【実習】μT-Kernel 入門
(協力：ルネサス エレクトロニクス)

μT-Kernel 入門

TRON Forum

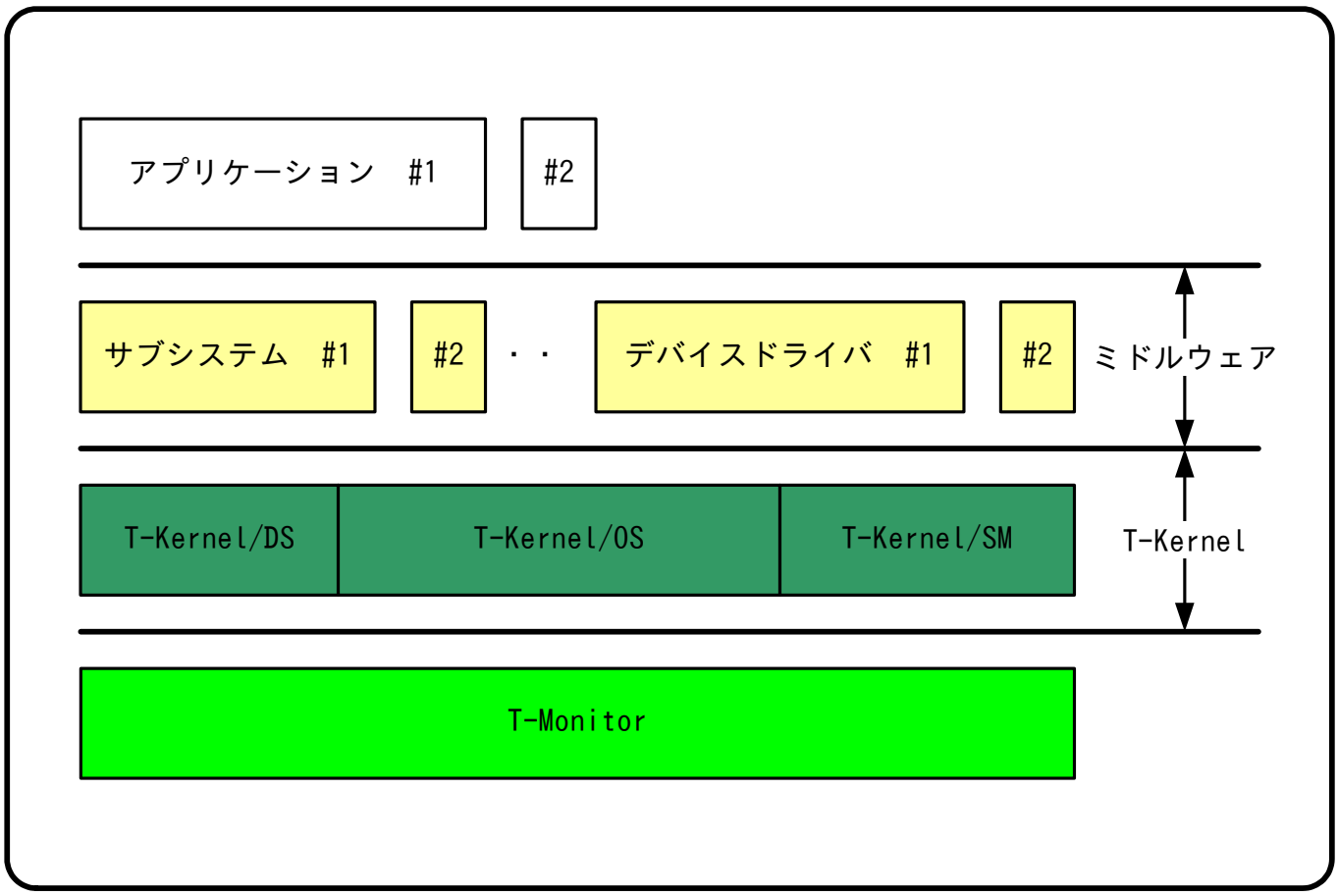
目次

第１章

μT-Kernel の概要と仕様の概念

1.1　μT-Kernelの概要

T-Kernelの概要



T-Kernelの概要

T-Kernel は、T-Kernel/OperatingSystem（T-Kernel/OS）、T-Kernel/SystemManager（T-Kernel/SM）、および T-Kernel/DebuggerSupport（T-Kernel/DS）を含む全体を指すが、T-Kernel/OSのみを（狭義の）T-Kernelと呼ぶ場合もある。

T-Kernel/OperatingSystem（T-Kernel/OS）は以下のような機能を提供する。

タスク制御機能、タスク間同期通信機能、メモリ管理機能
例外/割込み制御機能、時間管理機能、サブシステム管理機能

T-Kernel/SystemManager（T-Kernel/SM）は以下のような機能を提供する。

システムメモリ管理機能、アドレス空間管理機能、デバイス管理機能、割込み管理機能
I/Oポートアクセスサポート機能、省電力機能、システム構成情報管理機能

T-Kernel/DebuggerSupport（T-Kernel/DS）はデバッガ専用に以下のような機能を提供する。

カーネルの内部状態の参照、実行のトレース

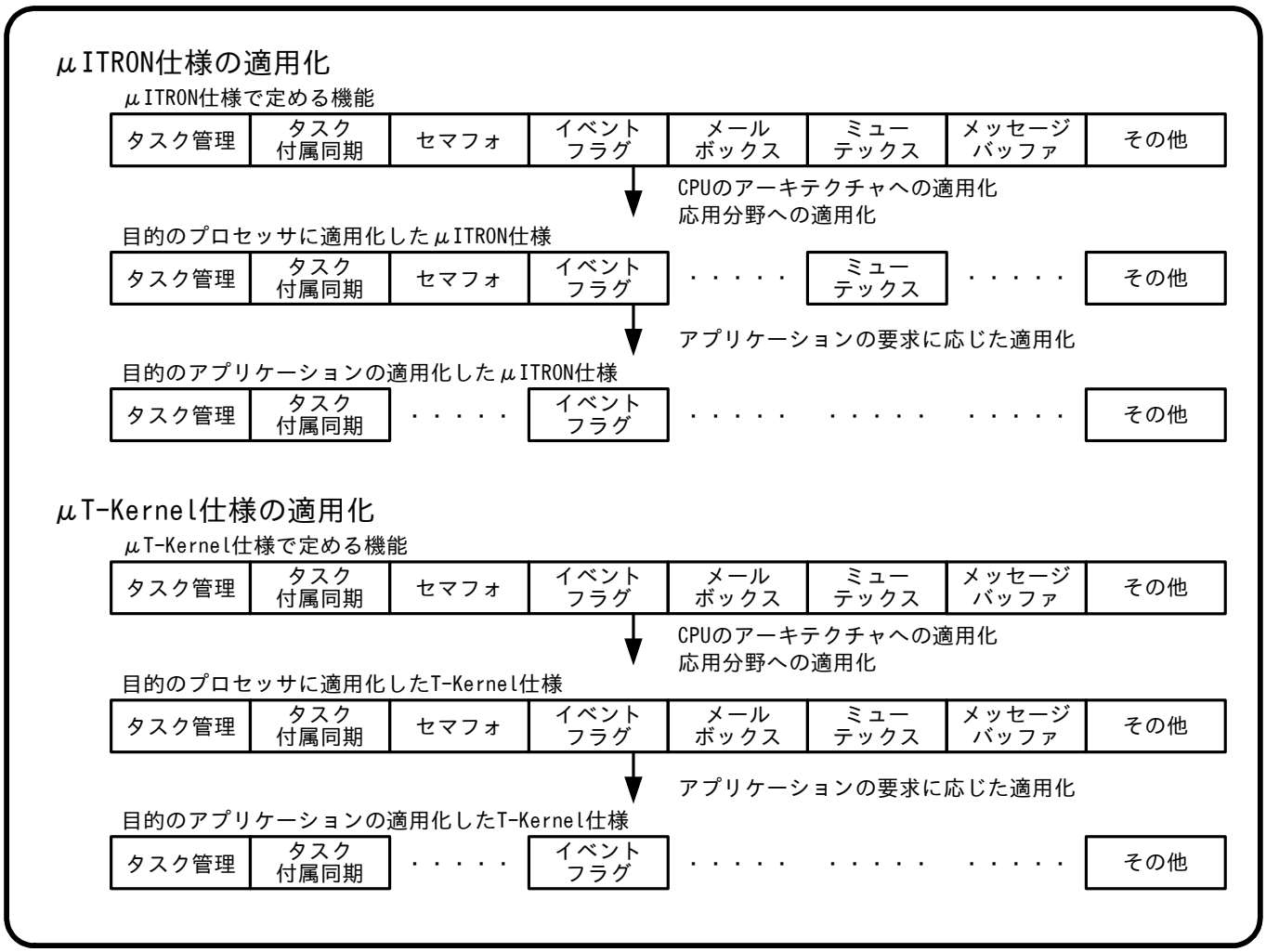
μT-Kernelの概要

一方、μT-Kernelは小規模組み込みシステム向けの最適化・適応化を行ないやすい設計にするという方針の下に仕様が策定されている。なおかつ、T-Kernelとの互換性も考慮に入れ、デバイスドライバやミドルウェアの流通性や移植性の向上を狙っている。

【T-Kernelとの差異】

μT-Kernelには保護レベルが無いため、T-Kernel/OS、T-Kernel/SMといった切り分けも存在しない。
また、T-Kernel/DSという分類もなく、同等の機能はデバッガサポート機能として定められている。
その結果、μT-Kernelには以下の機能が存在しない。

タスク例外機能、システムメモリ管理機能、アドレス空間管理機能
I/Oポートアクセスサポート機能、省電力管理機能、システム構成情報管理機能



T-Kernel仕様の適用化

T-Kernelは、組み込みシステム用のリアルタイムカーネルとして、小規模から大規模のさまざまなシステムをターゲットとしており、デバイスドライバやミドルウェアの流通性の向上をねらっています。このため、T-Kernelは規模の大きなシステムにも対応できるように仕様が策定されています。そのため、小規模なシステム向けには必須とはされない機能も含まれていますが、サブセット仕様を定めてしまうとデバイスドライバやミドルウェア等の流通性や移植性を妨げることになります。

T-Kernelでは、レベル分けなどのサブセット化のための仕様は定めていません。原則として、すべてのT-Kernel仕様OSはすべての仕様を実装しなければなりません。ただし、ターゲットシステムにおいてハードウェア上の制約で実現不可能な機能は簡易な実装として構わないとされています。

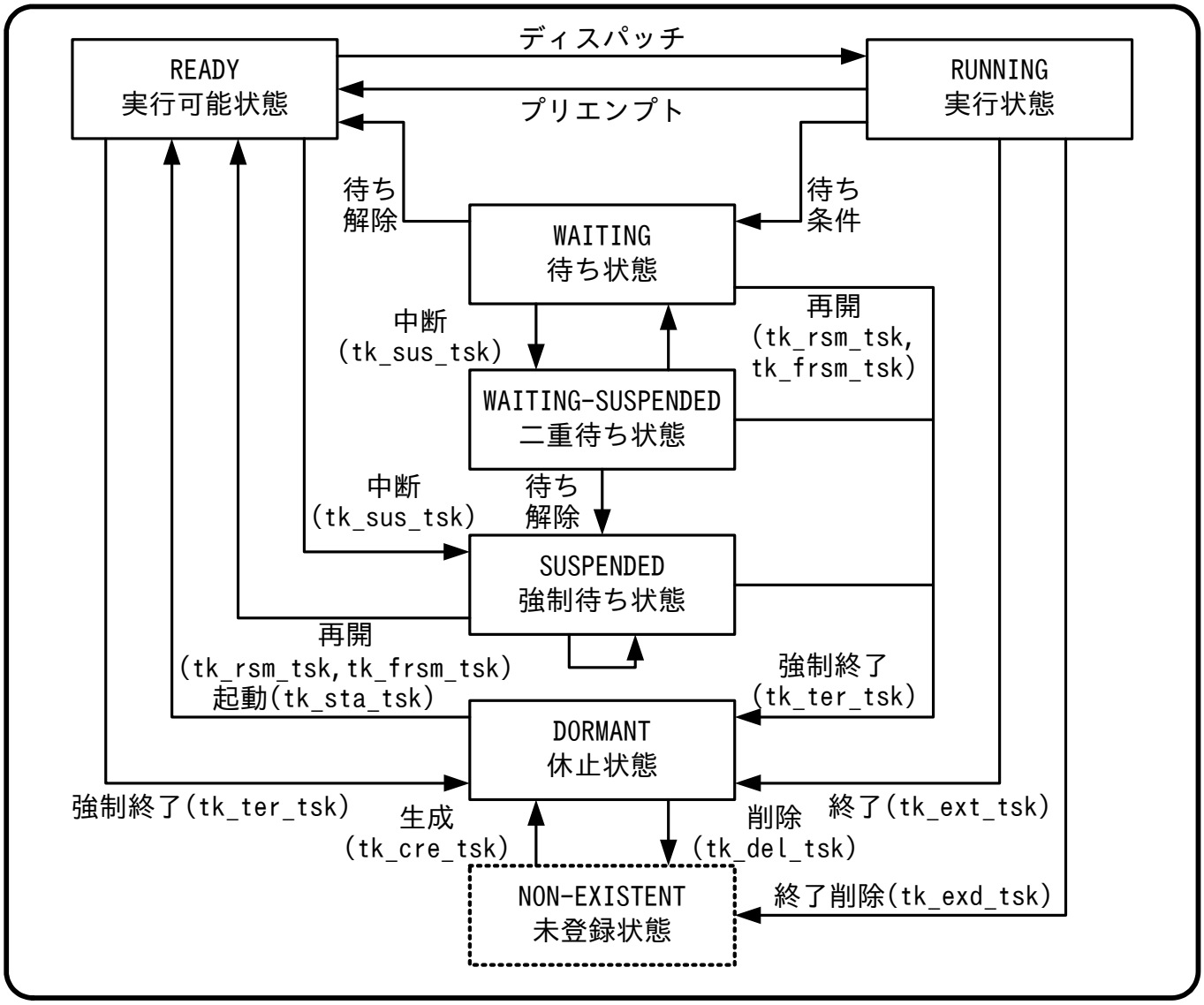
μT-Kernel仕様の適用化

μT-KernelはT-Kernelとの互換性を考慮に入れて仕様が策定されている。つまり、μT-KernelとT-Kernelの間でプログラムの移植を行う際、共通に存在する機能のみを使っていれば再コンパイルだけで移植できるように、また、修正が避けられない場合でもそれが最小限で済むように仕様が策定されています。そのため、μT-Kernel仕様には、あるシステムにおいては必要とされないような機能も含まれています。

μT-Kernelでは、レベル分けなどのサブセット化のための仕様は定めていません。全てのμT-Kernel仕様OSは全ての機能を実装しなければなりません。ただし、μT-Kernelをターゲットシステムに実装する際、必要とされない機能を省くことは構わないとされています。つまり、OS提供者は仕様に含まれる全ての機能を提供しなくてはならないが、利用者がその中から必要な機能だけを選んで使えるようにするのは構わないとされています。

1.2　μT-Kernel仕様の概念

タスクの状態遷移図



タスク状態

タスク状態は、大きく次の５つに分類されます。この内、広義の待ち状態は、さらに３つの状態に分類されます。また、実行状態と実行可能状態を総称して、実行できる状態と呼びます。これらタスクの状態はμITRON4.0仕様と同じですが、タスクの初期状態はNON-EXISTENTの未登録状態から常に始まります。μITRON4.0仕様のような静的APIは、T-Kernel仕様/μT-Kernel仕様には存在しません。

(1) 実行状態（RUNNING）
現在そのタスクを実行中であるという状態。ただし、タスク独立部を実行している間は、別に規定されている場合を除いて、タスク独立部の実行を開始する前に実行していたタスクが実行状態であるものとします。

(2) 実行可能状態（READY）
そのタスクを実行する準備は整っているが、そのタスクよりも優先順位の高いタスクが実行中であるために、そのタスクを実行できない状態。言い換えると、実行できる状態のタスクの中で最高の優先順位になればいつでも実行できる状態。

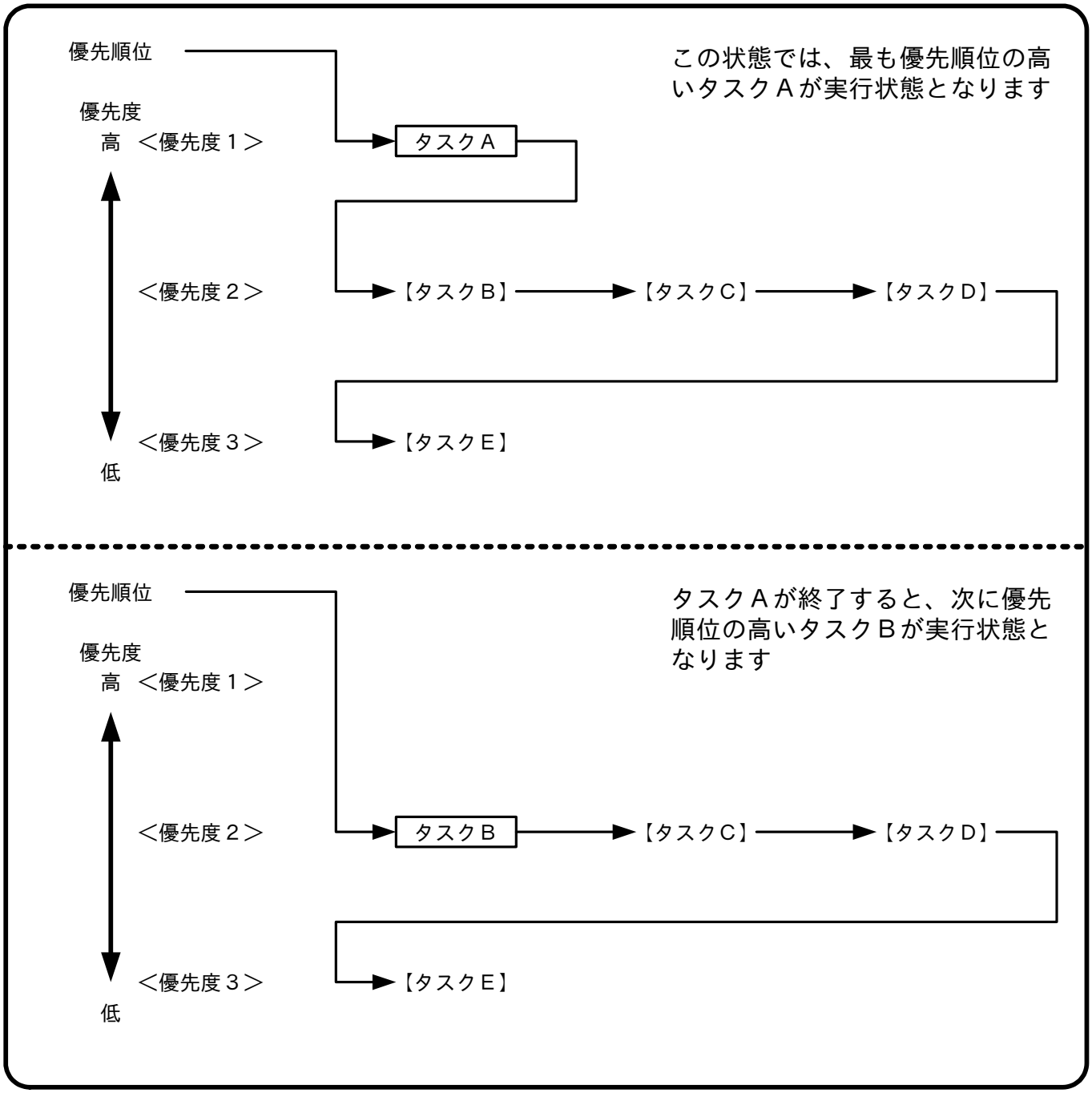
(3) 広義の待ち状態
そのタスクを実行できる条件が整わないために実行ができない状態。言い換えると、何らかの条件が満たされるのを待っている状態。タスクが広義の待ち状態にある間、プログラムカウンタやレジスタなどのプログラムの実行状態を表現する情報は保存されています。タスクを広義の待ち状態から実行再開する時には、プログラムカウンタやレジスタなどを広義の待ち状態になる直前の値に戻します。広義の待ち状態は、さらに次の３つの状態に分類されます。
(3.1) 待ち状態（WAITING）
何らかの条件が整うまで自タスクの実行を中断するシステムコールを呼出したことにより、実行が中断された状態。
(3.2) 強制待ち状態（SUSPENDED）
他のタスクによって、強制的に実行を中断させられた状態。
(3.3) 二重待ち状態（WAITING-SUSPENDED）
待ち状態と強制待ち状態が重なった状態。待ち状態にあるタスクに対して、強制待ち状態への移行が要求されると、二重待ち状態に移行されます。
μT-Kernelでは「待ち状態(WAITING)」と「強制待ち状態(SUSPENDED)」を明確に区別しており、タスクが自ら強制待ち状態(SUSPENDED)になることはできません。

(4) 休止状態（DORMANT）
タスクがまだ起動されていないか、実行を終了した後の状態。タスクが休止状態にある間は、実行状態を表現する情報は保存されていません。タスクを休止状態から起動する時には、タスクの起動番地から実行を開始します。また、別に規定されている場合を除いて、レジスタの内容は保証されません。

(5) 未登録状態（NON-EXISTENT）
タスクがまだ生成されていないか、削除された後の、システムに登録されていない仮想的な状態。

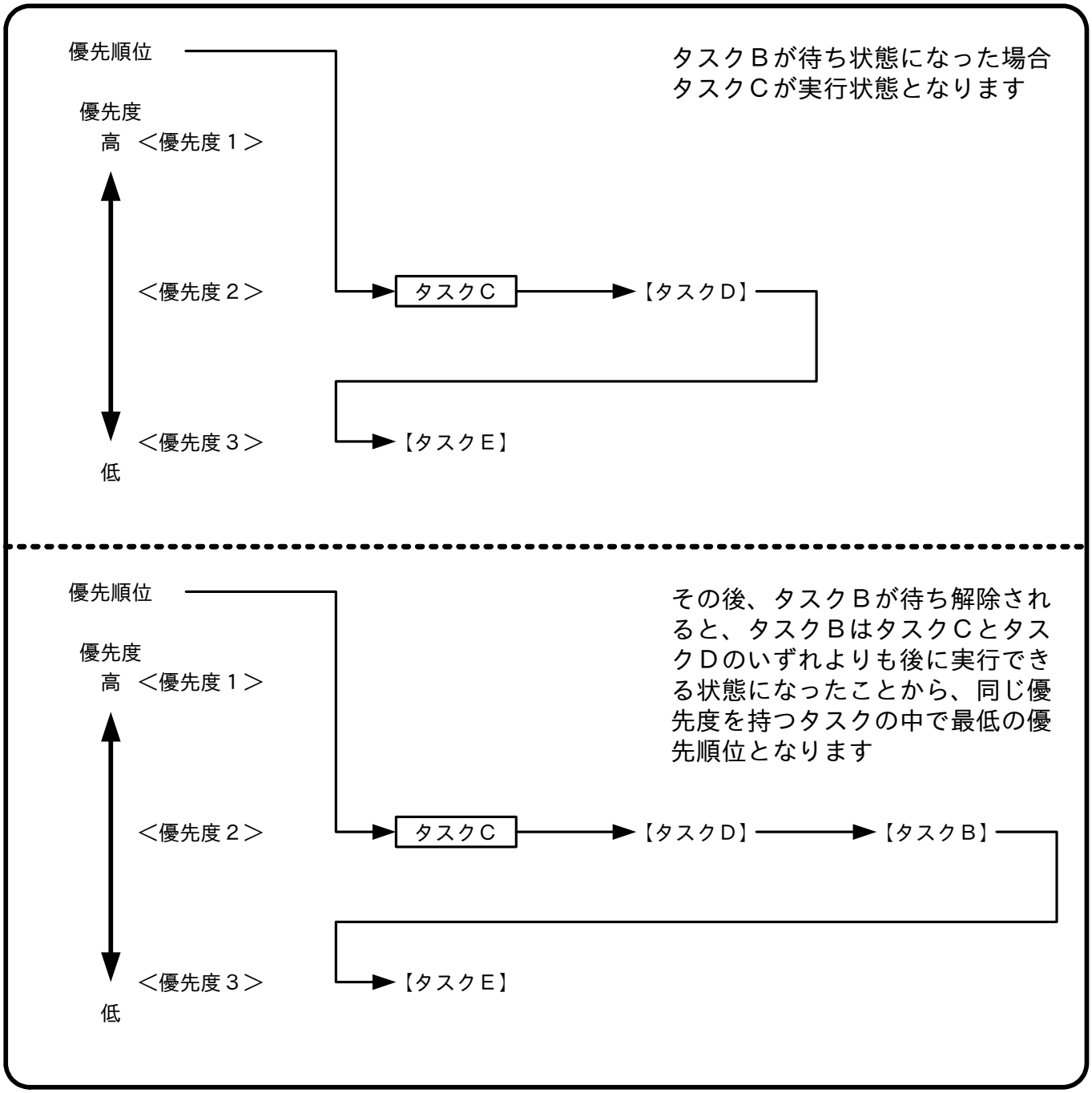
実行可能状態に移行したタスクが、現在実行中のタスクよりも高い優先順位を持つ場合には、実行可能状態への移行と同時にディスパッチが起こり、即座に実行状態へ移行する場合があります。この場合、それまで実行状態であったタスクは、新たに実行状態へ移行したタスクにプリエンプトされたと言います。また、システムコールの機能説明などで、「実行可能状態に移行させる」と記述されている場合でも、タスクの優先順位によっては、即座に実行状態に移行させる場合もあります。
タスクの起動とは、休止状態のタスクを実行可能状態に移行させることを言います。このことから、休止状態と未登録状態以外の状態を総称して、起動された状態と呼ぶことがあります。タスクの終了とは、起動された状態のタスクを休止状態に移行させることを言います。
タスクの待ち解除とは、タスクが待ち状態の時は実行可能状態に、二重待ち状態の時は強制待ち状態に移行させることを言います。また、タスクの強制待ちからの再開とは、タスクが強制待ち状態の時は実行可能状態に、二重待ち状態の時は待ち状態に移行させることを言います。

タスクのスケジューリング規則



タスクのスケジューリング規則

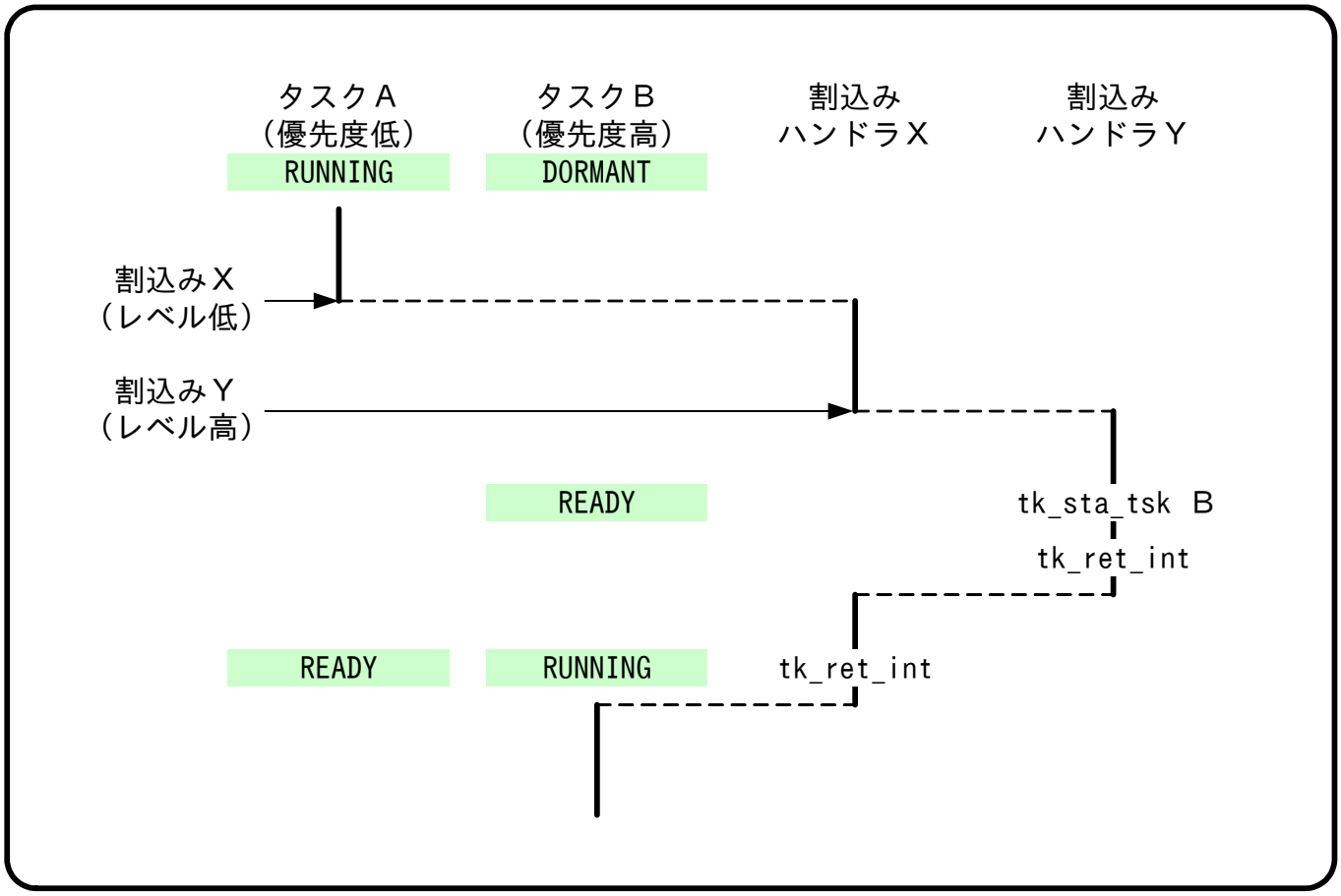
μT-Kernel仕様においては、タスクに与えられた優先度に基づくプリエンプティブな優先度ベーススケジューリング方式を採用しています。同じ優先度を持つタスク間では、FCFS（First Come First Served）方式によりスケジューリングを行います。具体的には、タスクのスケジューリング規則はタスク間の優先順位を用いて、タスク間の優先順位はタスクの優先度によって、それぞれ次のように規定されます。実行できるタスクが複数ある場合には、その中で最も優先順位の高いタスクが実行状態となり、他は実行可能状態となる。タスク間の優先順位は、異なる優先度を持つタスク間では、高い優先度を持つタスクの方が高い優先順位を持ちます。同じ優先度を持つタスク間では、先に実行できる状態（実行状態または実行可能状態）になったタスクの方が高い優先順位を持ちます。ただし、システムコールの呼出しにより、同じ優先度を持つタスク間の優先順位が変更される場合があります。



以上を整理すると、実行可能状態のタスクが実行状態になった後に実行可能状態に戻った直後には、同じ優先度を持つタスクの中で最高の優先順位を持っているのに対して、実行状態のタスクが待ち状態になった後に待ち解除されて実行できる状態になった直後には、同じ優先度を持つタスクの中で最低の優先順位となります。

最も高い優先順位を持つタスクが替わった場合には、ただちにディスパッチが起こり、実行状態のタスクが切り替わります。ただし、ディスパッチが起こらない状態になっている場合には、実行状態のタスクの切替えは、ディスパッチが起こる状態となるまで保留されます。

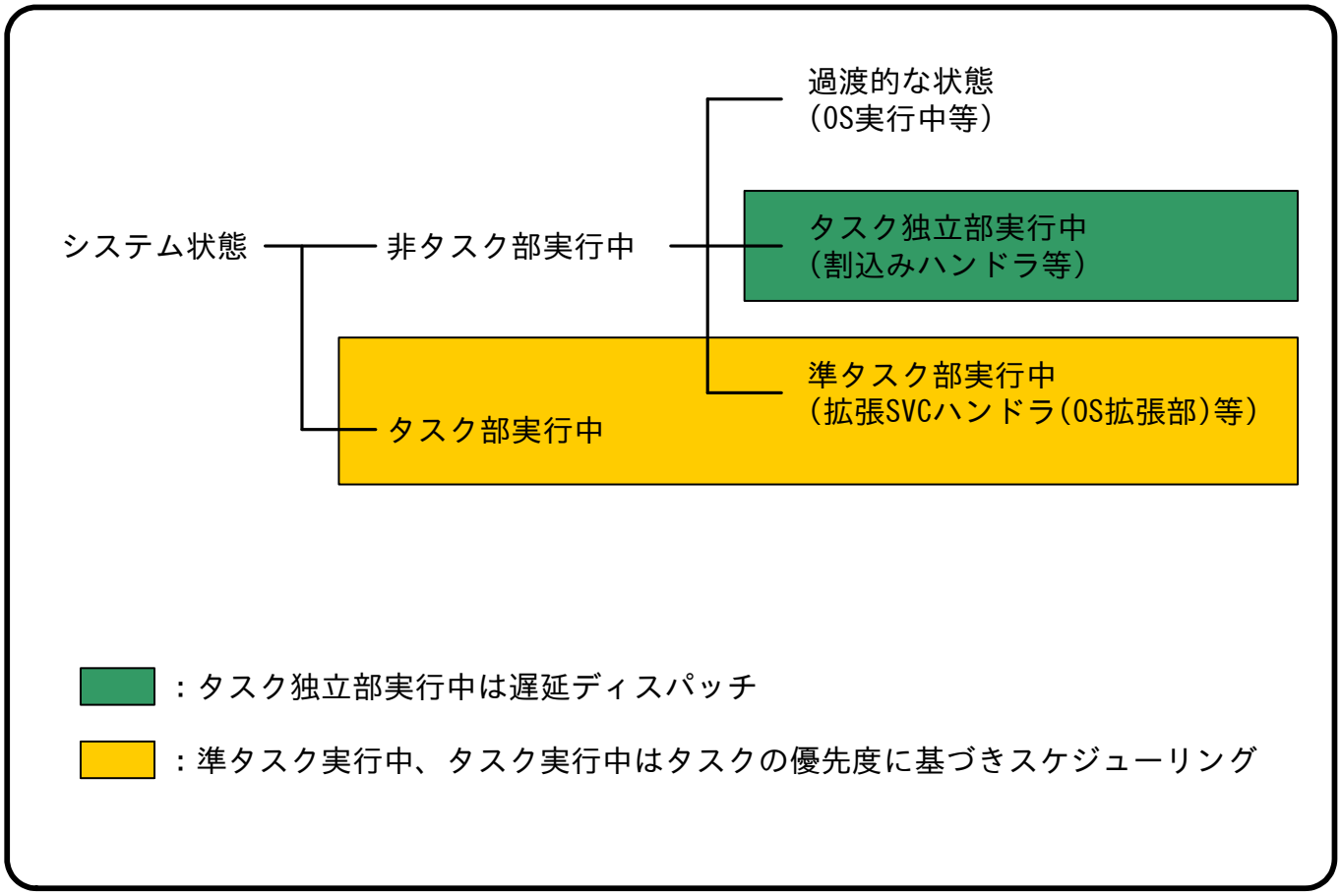
割込みのネストと遅延ディスパッチ



割込みのネストと遅延ディスパッチ

タスク A の実行中に割込み X が発生し、その割込みハンドラ中でさらに高優先度の割込み Y が発生した状態を示しています。この場合、割込み Y からのリターン時に即座にディスパッチを起こしてタスク B を起動すると、割込み X の残りの部分の実行がタスク B よりも後回しになり、タスク A が実行状態になった時に、はじめて割込み X の残りの部分が実行されることになります。これではレベルの低い割込み X のハンドラが、レベルの高い割込みばかりではなく、それによって起動されたタスク B にもプリエンプトされる危険を持つことになります。また、割込みハンドラがタスクに優先して実行されるという保証がなくなり、割込みハンドラが書けなくなってしまいます。遅延ディスパッチの原則を設けているのは、こういった理由によるものです。

システム状態



システム状態

システム状態は、上記のように分類されており、「過渡的な状態」は、OS 実行中（システムコール実行中）の状態に相当します。ユーザから見ると、ユーザの発行したそれぞれのシステムコールが不可分に実行されるということが重要なのであり、システムコール実行中の内部状態はユーザからは見えません。OS実行中の状態を「過渡的な状態」と考え、その内部をブラックボックス的に扱ってください。

しかし、次の場合、過渡的な状態が不可分に実行されません。

- メモリの獲得・解放を伴うシステムコールで、メモリの獲得・解放を行っている間。

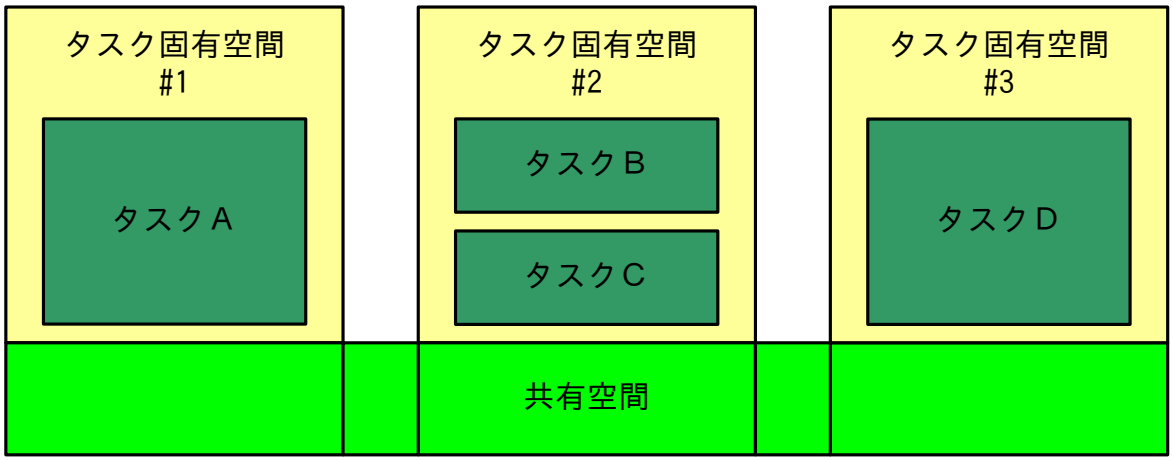
このような、過渡的な状態にあるタスクに対して、タスクの強制終了(tk_ter_tsk)を行った場合の動作は保証されません。また、タスクの強制待ち(tk_sus_tsk)も過渡的な状態のまま停止することになり、それによりデッドロック等を引き起こす可能性があります。

したがって、tk_ter_tsk、tk_sus_tskは原則として使用できません。これらを使用するのは、デバッガのようなOSに極近い、OSの一部といえるようなサブシステム内のみとするべきです。

「タスク独立部」、「準タスク部」は、ハンドラ実行中の状態に相当します。ハンドラのうち、タスクのコンテキストを持つ部分が「準タスク部」であり、タスクとは独立したコンテキストを持つ部分が「タスク独立部」です。具体的には、ユーザの定義した拡張システムコールの処理を行う拡張 SVC ハンドラが「準タスク部」であり、外部割込みによって起動される割込みハンドラやタイムイベントハンドラが「タスク独立部」です。「準タスク部」では、一般のタスクと同じようにタスクの状態遷移を考えることができ、待ち状態に入るシステムコールも発行可能です。

「過渡的な状態」、「タスク独立部」、「準タスク部」を合わせて「非タスク部」と呼びます。これ以外で、普通にタスクのプログラムを実行している状態が「タスク部実行中」の状態です。

メモリ空間と保護レベル



システムアプリケーション

	保護レベル	用途
高	レベル０	システムソフトウエア(OS、デバイスドライバ等)
↕	レベル１	システムアプリケーション
↕	レベル２	(予約)
低	レベル３	ユーザアプリケーション

T-Kernel仕様におけるメモリ

T-Kernel仕様では、メモリのアドレス空間は共有空間とタスク固有空間に区別されます。共有空間はすべてのタスクから同じようにアクセスできる空間であり、タスク固有空間はそのタスク固有空間に属しているタスクからのみアクセスできる空間です。１つのタスク固有空間に複数のタスクが属している場合もあります。

タスク固有空間と共有空間の論理アドレス空間は、CPU(およびMMU)の制限に依存するために実装依存となりますが、タスク固有空間を低位アドレス、共有空間を高位アドレスにおくことを原則とします。また、メモリには常駐メモリと非常駐メモリがあります。

また、T-Kernel仕様には４段階の保護レベルがあり、現在実行中の保護レベルと同じか低い保護レベルのメモリにのみアクセスできるようになっています。非タスク部(タスク独立部、準タスク部など)は保護レベル０で実行されます。保護レベル１～３で実行できるのはタスク部のみであり、保護レベル０でもタスク部は実行可能となっています。

【T-Kernel仕様との差異】

μT-Kernel 仕様にはタスク固有空間は存在せず、T-Kernel 仕様の共有空間に相当するものしか存在しません。同様に μT-Kernel仕様にはT-Kernel仕様の常駐メモリに相当するものしか存在しません。

また、μT-Kernel 仕様でも０～３の４段階の保護レベルが指定できますが、MMU の無いシステムを前提としているため、どの保護レベルを指定されても保護レベル０として扱うものとします。

μT-Kernel仕様に保護レベルの指定が存在するのは、T-Kernel仕様との互換性を保つためであり、μT-Kernelだけで使う場合は保護レベル０を指定しておけば問題ありません。もし、μT-Kernel仕様とT-Kernel仕様のどちらでも使用するようなアプリケーションを作成したい場合は、T-Kernel仕様での保護レベルに合わせた指定をしておけば良いことになります。

第 2 章

μT-Kernel 仕様共通規定

2.1　データ型

汎用的なデータ型

■ μT-Kernel仕様はμITRON仕様と同じく専用のデータ型を使用
- 内容的にはμITRON仕様とほぼ同等
- 「<tk/tkernel.h> 」をインクルードすることで使用可能

例：
```
typedef signed char    B;       /* 符号付き  8ビット整数                */
typedef signed short   H;       /* 符号付き 16ビット整数                */
typedef signed long    W;       /* 符号付き 32ビット整数                */
typedef unsigned char  UB;      /* 符号無し  8ビット整数                */
typedef unsigned short UH;      /* 符号無し 16ビット整数                */
typedef unsigned long  UW;      /* 符号無し 32ビット整数                */
typedef signed int     INT;     /* プロセッサのビット幅の符号付き整数   */
typedef unsigned int   UINT;    /* プロセッサのビット幅の符号無し整数   */

#define LOCAL  static           /* ローカルシンボル定義                 */
#define EXPORT                  /* グローバルシンボル定義               */
#define IMPORT extern           /* グローバルシンボル参照               */
```

■ その他のヘッダファイル
- 各CPU毎の共通基本データ型　→　「<basic.h> 」
- T-Monitorのヘッダファイル　→　「<tk/tmonitor.h> 」

汎用的なデータ型

μT-Kernel仕様でも、μITRON仕様やT-Kernel仕様と同様にシステムコールで使用する汎用的なデータ型にはマクロ名が定義されています。

stksz、wupcnt、メッセージサイズなど、明らかに負の数にならないパラメータも、原則として符号付き整数(INT、W)のデータ型となっています。これは、整数はできるだけ符号付きの数として扱うというTRON全般のルールに基づいたものです。また、タイムアウト(TMO tmout)のパラメータでは、これが符号付きの整数であることを利用し、TMO_FEVR(=-1)を特殊な意味に使っています。符号無しのデータ型を持つパラメータは、ビットパターンとして扱われるもの(オブジェクト属性やイベントフラグなど)があります。

特殊なデータ型

■ 出現頻度の高いデータ型や特殊な意味を持つデータ型

例：
```
typedef INT    FN;                  /* 機能コード                    */
typedef INT    RNO;                 /* ランデブ番号                  */
typedef UW     ATR;                 /* オブジェクト/ハンドラ属性     */
typedef INT    ER;                  /* エラーコード                  */
typedef INT    PRI;                 /* 優先度                        */
typedef W      TMO;                 /* タイムアウト指定              */
typedef UW     RELTIM;              /* 相対時間                      */
typedef struct systim {             /* システム時刻                  */
        W      hi;                  /* 上位32 ビット                 */
        UW     lo;                  /* 下位32 ビット                 */
} SYSTIM;

#define NULL          0             /* 無効ポインタ                  */
#define TA_NULL       0             /* 特別な属性を指定しない        */
#define TMO_POL       0             /* ポーリング                    */
#define TMO_FEVR      (-1)          /* 永久待ち                      */
```

特殊なデータ型

パラメータの意味を明確にするため、出現頻度の高いデータ型や特殊な意味を持つデータ型に対しては、上記のような名称を使用します。

複数のデータ型を複合したデータ型の場合は、その内の最も主となるデータ型で代表します。例えば、tk_cre_tskの戻値はタスクIDかエラーコードですが、主となるのはタスクIDであるため、データ型はIDとなります。

【T-Kernel仕様1.00.00との差異】

符号付き型の定義でsignedが明示的に指定されています。また、32ビット型の定義がintではなくlongで行われています。いくつかの型定義でINTがWに、UINTがUWになっています。これは32ビット幅であるべき型はW、UWとすべきという考えからです。

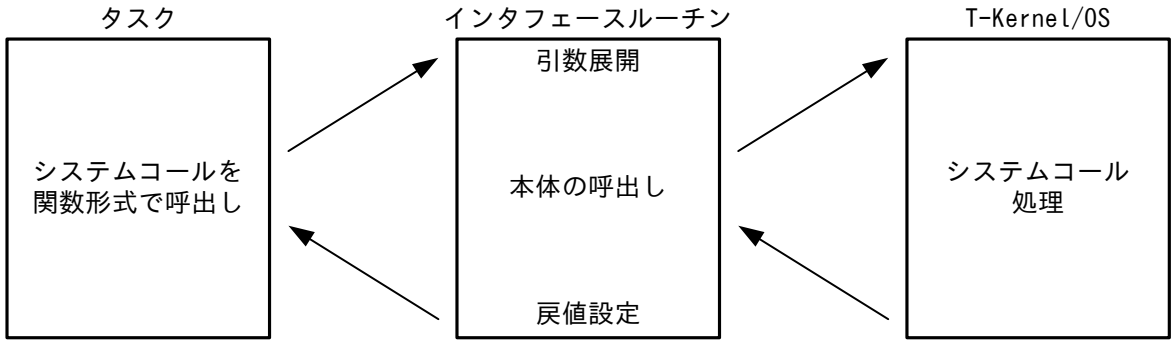
2.2　システムコール

システムコールの形式（T-Kernel仕様）

■ C言語からシステムコールを実行する場合のインタフェース方法を標準化
- ● システムコールはすべてC言語の関数として定義
- ● 関数としての戻り値は 0 または正の値が正常終了、負の値がエラーコード

■ C言語のシステムコールインタフェース方法
- ● T-Kernel仕様ではライブラリで提供することを規定
- ● μT-Kernel仕様では実装定義（マクロ/インライン関数/インラインアセンブラ等）

■ アセンブラレベルのインタフェース方法は実装定義



システムコールの形式

システムコールの形式

μT-Kernel仕様では、C言語を標準的な高級言語として使用することにしており、C言語からシステムコールを実行する場合のインタフェース方法を標準化しています。

一方、アセンブラレベルのインタフェース方法は実装定義となっています。アセンブラでプログラムを作成する場合でも、C 言語のインタフェースを利用して呼び出す方法を推奨しています。理由は、アセンブラで作成したプログラムも同一CPUであればOSが変わっても移植性が確保できるからです。

システムコールインタフェースでは、次のような共通原則を設けています。

- システムコールはすべてC言語の関数として定義されています。
- 関数としての戻り値は、0 または正の値が正常終了、負の値がエラーコードとなっています。

なお、μT-Kernel 仕様ではシステムコールインタフェースの実装方法は実装定義となっています。例えば、C言語のマクロやインライン関数、インラインアセンブラなどを用いて実装することが許されています。

【T-Kernel仕様との差異】

T-Kernel仕様ではシステムコールインタフェースはライブラリとすると定められていますが、μT-Kernel仕様では実装定義となっています。これはμT-Kernel仕様では異なるコンパイラ間での移植性よりも実行効率を優先する方針をとっているためです。

RX62N用にカスタマイズされたμT-Kernelでも、システムコールの呼出し側はC言語の関数の呼出形式で直接呼出します。インタフェースライブラリを経由したシステムコールの呼び出しはサポートしていません。

タスク独立部およびディスパッチ禁止状態から発行できるシステムコール

■ タスク独立部およびディスパッチ禁止状態から発行可能なシステムコール
- tk_sta_tsk タスクの起動
- tk_wup_tsk タスクの起床
- tk_rel_wai タスク待ちの強制解除
- tk_sus_tsk タスクの強制待ち
- tk_sig_sem セマフォへの資源返却
- tk_set_flg イベントフラグのセット
- tk_rot_rdq タスクの優先順位の回転
- tk_get_tid 実行状態タスクのID 参照
- tk_sta_cyc 周期ハンドラの動作開始
- tk_stp_cyc 周期ハンドラの動作停止
- tk_sta_alm アラームハンドラの動作開始
- tk_stp_alm アラームハンドラの動作停止
- tk_ref_tsk タスク状態の参照
- tk_ref_cyc 周期ハンドラ状態の参照
- tk_ref_alm アラームハンドラ状態の参照
- tk_ref_sys システム状態の参照
- tk_ret_int 割込みハンドラからの復帰

■ ディスパッチ禁止状態/可能状態、どちらでも同一動作を行うシステムコール
- tk_fwd_por ランデブポートに対するランデブ回送
- tk_rpl_rdv ランデブ返答

上記以外のシステムコールをタスク独立部およびディスパッチ禁止状態から発行した場合の動作は実装定義となります。

【T-Kernel仕様との差異】

タスク独立部およびディスパッチ禁止状態から発行できなくてはならないシステムコールの一覧にtk_sig_tevは存在しません。これは、μT-Kernel仕様には、tk_sig_tevシステムコールが存在しないためです。

システムコールの呼出し制限（T-Kernel仕様）

■ システムコールを呼び出すことのできる保護レベルを制限可能
■ ただし、拡張SVCの呼出しは制限されない
- 指定した保護レベルより低い保護レベルから呼び出した場合、E_QACVとなる
- そのようなタスクはサブシステムの機能を使ってプログラムする
- システムコールの呼出し保護レベルはシステム構成情報(TSVCLimit)として定義

■メモリの保護レベル
- メモリ領域には保護レベルが設定される
- 各タスクには保護レベルに対応した実行レベルが設定され、実行レベルに合わせてメモリアクセスを管理する
- 保護レベルが低いと共有空間上のメモリ領域であっても、アクセスできない

高 ↕ 低

保護レベル	用途
レベル０	システムソフトウエア(OS、デバイスドライバ等)
レベル１	システムアプリケーション
レベル２	(予約)
レベル３	ユーザアプリケーション

μT-Kernel仕様では全て保護レベル0として扱う

システムコールの呼出し制限

T-Kernel仕様では、システムコールを呼び出すことのできる保護レベルを設定することが可能です。この場合、指定した保護レベルより低い保護レベルで動作しているタスク（タスク部）からシステムコールを発行した場合、E_OACVエラーとなります（拡張SVCの呼出しは制限されません）。

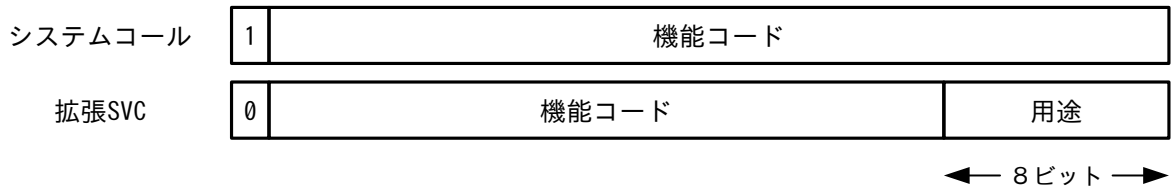
例えば、保護レベル1より低い保護レベルからのシステムコールの呼出しを禁止した場合、保護レベル2、3で実行しているタスクからはシステムコールは発行できなくなります。つまり、保護レベル2、3で動作しているタスクは、拡張 SVC しか発行できないということになり、サブシステムの機能のみを使ってプログラムすることになります。

これはT-KernelをT-Kernel Extensionと組み合わせる場合、T-Kernel Extensionの仕様に基づくタスクから、T-Kernelの機能を直接操作させないために使用します。システムコール呼出し制限をする保護レベルは、システム構成情報管理機能により設定します。

一方、μT-Kernel仕様はMMUのないシステムを前提としており、保護レベルの機能はサポートしていない（全て保護レベル0として扱う）ため、保護レベルによるシステムコール呼び出しの制限はありません。

機能コード（T-Kernel仕様）

■ システムコールを識別するために各システムコール毎に割り付ける番号
- システムコールは負の値(＜0)、拡張SVCは正の値(≧0)を使用する
- 拡張SVCの場合、下位８ビットはサブシステムID、上位ビットが機能コードとなる



■ μT-Kernel仕様ではシステムコールインタフェースは実装定義
■ 機能コードを用いない形式で呼び出す実装も有り得る

機能コード

機能コードとは、システムコールを識別するために各システムコールに割り付けられる番号のことです。

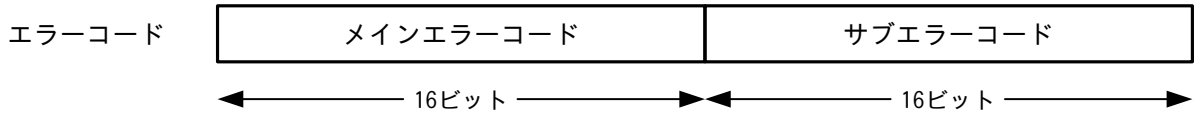
T-Kernel仕様ではシステムコールの機能コードは特に定めていません。すべて実装定義となっていますが、現状は上記のような実装となっています。

それはμT-Kernel仕様も同様であり、システムコールの機能コードの具体的な値は実装定義ですが、システムコール毎にそれぞれ異なる負の値を割り付けることとなっています。しかも、μT-Kernel仕様ではシステムコールインタフェースも実装定義であり、機能コードを用いない形式で呼び出す実装も有り得ます。

エラーコード（T-Kernel仕様）

■ システムコールの戻値はエラーコードであり、原則として符号付き整数値
■ エラーが発生した場合は負の値となる
■ 処理を正常に終了した場合はE_OK(=0)または正の値となる
　● エラーコードはメインエラーコードとサブエラーコードで構成
　● 下位16ビットがサブエラーコード、上位16ビットがメインエラーコードとなる
　● T-Kernel/OSではサブエラーコードは使用せず、常にゼロとなる

```
#define MERCD(er)      ( (ER)(er) >> 16 )                  /* メインエラーコード */
#define SERCD(er)      ( (H)(er) )                         /* サブエラーコード   */
#define ERCD(mer,ser) ( (ER)(mer) << 16 | (ER)(UH)(ser) )
```



μT-Kernel仕様ではメインエラーコードのみを扱う

エラーコード

システムコールの戻値は原則として符号付きの整数であり、エラーが発生した場合には負の値のエラーコード、処理を正常に終了した場合はE_OK(=0)または正の値となります。正常終了した場合の戻値の意味はシステムコール毎に規定されています。この原則の例外は呼び出されるとリターンすることのないシステムコールです。リターンすることのないシステムコールは、C言語APIでは戻値を持たないもの（すなわちvoid 型の関数）として宣言されています。

また、T-Kernel 仕様において、エラーコードはメインエラーコードとサブエラーコードで構成されています。エラーコードの下位16ビットがサブエラーコード、残りの上位ビットがメインエラーコードです。メインエラーコードは、検出の必要性や発生状況などにより、エラークラスに分類されています。T-Kernel/OSではサブエラーコードは使用せず、常に0となります。

【T-Kernel仕様との差異】

T-Kernel仕様では下位16ビットがサブエラーコード、上位16ビットがメインエラーコードと定められていますが、μT-Kernel仕様ではメインエラーコードのみとなっています。例えばE_IDのメインエラーコードは-18と定義されていますが、T-Kernel仕様では上位16ビットにこの値が入り、下位16ビットには0が入ります。

一方、μT-Kernel仕様ではE_IDは-18と定義されています。互換性確保のために、エラーコードは値を直接使うのではなくE_IDのようなマクロを使用しなければなりません。

T-Kernel仕様にはERCD、MERCD、SERCDといったエラーコード処理の為のマクロが存在しますが、μT-Kernel仕様には、これらは存在しません。これらのマクロを使用しているアプリケーションをμT-Kernel仕様に移植する場合、これらのマクロを使用しないように修正するか、同じ名前のマクロを定義する必要があります。

μT-Kernelのシステムコール概要

■ μT-Kernel仕様はμITRON4.0仕様(3.0仕様)とほぼ同等の機能を持つ
- ただし、システムコールレベルの細部仕様は異なる
- 互換性を保証している訳ではない

例：μT-Kernelのシステムコールは先頭に「tk_ 」が付いた名称となる

□ μITRON4.0仕様のsta_tskサービスコール
```
ER ercd = sta_tsk( ID tskid, VP_INT stacd );
```

□ μT-Kernel仕様のsta_tskシステムコール
```
ER ercd = tk_sta_tsk( ID tskid, INT stacd );
```

■ μT-Kernel仕様には存在しないμITRON4.0仕様の機能
- データキュー、オーバランハンドラ

μT-Kernelのシステムコール概要

μT-Kernel仕様のシステムコールはμITRON4.0仕様(3.0仕様)とほぼ同等の機能を持っています。ただし、システムコールレベルの細部仕様は異なっていますから、互換性を保証している訳ではありません。また、μITRON4.0仕様におけるデータキューとオーバランハンドラの機能はサポートしていません。

μT-Kernel仕様のシステムコール名称は、μITRON4.0仕様のサービスコール名称から考えると先頭に「 tk_ 」が付いた名称となります。ほぼ同等の名称であるため、μITRON仕様に慣れていれば違和感なくμT-Kernel仕様のシステムコールが記述可能です。

μITRON4.0仕様との相違点（１）

■ μT-Kernel仕様ではオブジェクトは全てカーネルによる自動割り付け
- μITRON4.0仕様における「cre_*** 」のサービスコールは廃止
- 「tk_cre_*** 」がμITRON4.0仕様の「acre_*** 」に相当

  tk_cre_tsk、tk_cre_sem、tk_cre_flg、tk_cre_mbx、tk_cre_mtx、tk_cre_mbf
  tk_cre_por、tk_cre_mpf、tk_cre_mpl、tk_cre_cyc、tk_cre_alm、tk_cre_res
- μT-Kernel仕様では静的APIはサポートしない

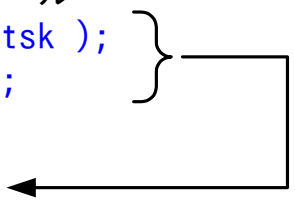
例：μT-Kernel仕様ではタスクの生成をtk_cre_tskシステムコールで行う

・μITRON4.0仕様のcre_tsk、acre_tskサービスコール

```
ER ercd = cre_tsk( ID tskid, T_CTSK *pk_ctsk );
ER_ID tskid = acre_tsk( T_CTSK *pk_ctsk );
```

・μT-Kernel仕様のtk_cre_tskシステムコール

```
ID tskid = tk_cre_tsk( T_CTSK *pk_ctsk );
```



μITRON4.0仕様との相違点（１）

μT-Kernel仕様では、カーネルオブジェクトは全てカーネルによるID番号の自動割り付けでしか生成できません。μITRON4.0仕様における静的APIはサポートしていません。ID番号を指定して生成する「cre_***」のサービスコール機能はなく、カーネルよる ID 番号の割り付けで生成を行う「acre_***」のサービスコール機能でのみカーネルオブジェクトの生成が可能です。

また、μT-Kernel仕様では「tk_cre_***」のシステムコールがμITRON4.0仕様における「acre_***」のサービスコールに相当します。パラメータパケットで生成情報を渡すことで対応したカーネルオブジェクトが生成されます。

μITRON4.0仕様との相違点（２）

■ μT-Kernel仕様では待ちを伴うシステムコールは一本化
- ●μITRON4.0仕様における「t***_*** 」のタイムアウト機能は廃止
- ●μITRON4.0仕様における「p***_*** 」のポーリング機能も廃止
- ●μT-Kernel仕様では待ちを伴うシステムコールが上記の機能を兼ねる
  tk_slp_tsk、tk_wai_tev、tk_wai_sem、tk_wai_flg、tk_rcv_mbx、tk_loc_mtx
  tk_snd_mbf、tk_rcv_mbf、tk_act_por、tk_get_mpf、tk_get_mpl
- ●タイムアウト指定はTMO_FEVR(永久待ち)で待ち方を指定
- ●ポーリング指定はTMO_POL(ポーリング)で待ち方を指定

例：μT-Kernel仕様ではメールボックスからの受信は
全てtk_rcv_mbxシステムコールで行う

・μITRON4.0仕様のrcv_mbxサービスコール

```
ER recd =  rcv_mbx( ID mbxid, T_MSG **ppk_msg );
ER recd = trcv_mbx( ID mbxid, T_MSG **ppk_msg, TMO tmout );
ER recd = prcv_mbx( ID mbxid, T_MSG **ppk_msg );
```

・μT-Kernel仕様のtk_rcv_mbxシステムコール

```
ER recd = tk_rcv_mbx( ID mbxid, T_MSG **ppk_msg, TMO tmout );
```

μITRON4.0仕様との相違点（２）

μT-Kernel仕様では、頭文字「t」のタイムアウト機能や頭文字「p」のポーリング機能のサービスコールは存在しません。タイムアウト機能やポーリング機能は WAITING 状態となる待ちを伴うシステムコールに集約されています。μT-Kernel仕様における待ちを伴うサービスコールには、タイムアウト指定のパラメータが存在し、そのパラメータが TMO_FEVR であれば永久待ち指定、TMO_POL であればポーリング指定、それ以外の正の値であればタイムアウト指定を行ったことになります。

メモ

第 3 章

μT-Kernel の機能

μT-Kernelの機能

■ タスク管理機能
■ タスク付属同期機能
■ 同期・通信機能
■ 拡張同期・通信機能
■ メモリプール管理機能
■ 時間管理機能
■ 割込み管理機能
■ システム状態管理機能
■ サブシステム管理機能
■ デバイス管理機能
■ デバッガサポート機能

デバッガサポート機能については、実装するかどうかはOS提供者の自由である。
ただし、実装する場合は仕様にあるインタフェースを守る必要がある。
また、デバッグに関する別の機能を提供する場合には、仕様にあるものと異なる名前を付けなければならない。

【T-Kernel仕様との差異】

μT-Kernel仕様には保護レベルが無いため、T-Kernel/OS、T-Kernel/SMといった切り分けも存在しません。また、T-Kernel/DSという分類もなく、同等の機能はデバッガサポート機能として定められています。

μT-Kernel仕様には以下の機能が存在しません。

- タスク例外機能
- システムメモリ管理機能
- アドレス空間管理機能
- I/Oポートアクセスサポート機能
- 省電力管理機能
- システム構成情報管理機能

また、以下の機能のうち一部のシステムコールが存在しません。

- タスク管理機能
- タスク付属同期機能
- システム状態管理機能
- サブシステム管理機能
- 割込み管理機能

どのシステムコールが存在しないかについては、個々の機能を参照してください。

逆に、T-Kernel仕様になくμT-Kernel仕様に存在する機能として、メモリ領域の引渡し機能があります。これは、タスクのスタックやメッセージバッファ・メモリプールで使用するメモリ領域として、ユーザーが確保した領域を指定する機能です。この機能により、スタックを高速な内蔵メモリに割り当てるなどの適応化が可能となるます。この機能を使わない場合、メモリ領域はOSが確保します。

具体的な使い方については、tk_cre_tsk、tk_cre_mbf、tk_cre_mpf、tk_cre_mplを参照してください。

3.1　タスク管理機能

タスク管理機能のシステムコール一覧

■タスク生成
```
ID tskid = tk_cre_tsk( T_CTSK *pk_ctsk );
```
■タスク削除
```
ER ercd  = tk_del_tsk( ID tskid );
```
■タスク起動
```
ER ercd  = tk_sta_tsk( ID tskid, INT stacd );
```
■自タスク終了
```
      void  tk_ext_tsk( void );
```
■自タスクの終了と削除
```
      void  tk_exd_tsk( void );
```
■他タスク強制終了
```
ER ercd  = tk_ter_tsk( ID tskid );
```
■タスク優先度変更
```
ER ercd  = tk_chg_pri( ID tskid, PRI tskpri );
```
■タスクレジスタの取得
```
ER ercd  = tk_get_reg( ID tskid, T_RFEGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
```
■タスクレジスタの設定
```
ER ercd  = tk_set_reg( ID tskid, T_RFEGS *pk_regs, T_EIT *pk_eit, T_CREGS *pk_cregs );
```
■タスク状態参照
```
ER ercd  = tk_ref_tsk( ID tskid, T_RTSK *pk_rtsk );
```
●stacdはINT型であり、処理系によって指定できる値の範囲が異なる（RX600用μT-Kernelは32ビット）

タスク管理機能は、タスクの状態を直接的に操作/参照するための機能です。タスクを生成/削除する機能、タスクを起動/終了する機能、タスクの優先度を変更する機能、タスクの状態を参照する機能が含まれます。タスクはID番号で識別されるオブジェクトであり、タスクのID番号をタスクIDと呼びます。タスク状態とスケジューリング規則については***を参照してください。

タスクは、実行順序を制御するためにベース優先度と現在優先度を持ちます。単にタスクの優先度といった場合には、タスクの現在優先度を指します。タスクのベース優先度は、タスクの起動時にタスクの起動時優先度に初期化されます。

ミューテックス機能を使わない場合には、タスクの現在優先度は常にベース優先度に一致しています。このため、タスク起動直後の現在優先度は、タスクの起動時優先度になっています。ミューテックス機能を使う場合に現在優先度がどのように設定されるかについては3-25頁で説明します。

カーネルは、タスクの終了時に、ミューテックスのロック解除を除いては、タスクが獲得した資源（セマフォ資源、メモリブロックなど）を解放する処理を行いません。タスク終了時に資源を解放するのは、アプリケーションの責任となります。

【T-Kernel仕様との差異】

以下のシステムコールはμT-Kernel仕様には存在しません。

- tk_chg_slt タスクスライスタイム変更
- tk_get_tsp タスク固有空間の参照
- tk_set_tsp タスク固有空間の設定
- tk_get_rid タスク所属リソースグループの取得
- tk_set_rid タスク所属リソースグループの設定
- tk_get_cpr コプロセッサのレジスタの取得
- tk_set_cpr コプロセッサのレジスタの設定
- tk_inf_tsk タスク統計情報参照

タスクの生成情報

pk_ctskの内容

```
typedef struct t_ctsk {
    VP    exinf;       /* Extended information     拡張情報                  */
    ATR   tskatr;      /* Task attribute           タスク属性                */
    FP    task;        /* Task startup address     タスク起動アドレス        */
    PRI   itskpri;     /* Priority at task startup タスク起動時優先度        */
    W     stksz;       /* User stack size (byte)   スタックサイズ(バイト数) */
    UB    dsname[8];   /* Object name              DSオブジェクト名称        */
    VP    bufptr;      /* User buffer              ユーザーバッファポインタ  */
} T_CTSK;
```

```
tskatr := (TA_ASM ∥ TA_HLNG) | [TA_USERBUF] | [TA_DSNAME] |
          (TA_RNG0 ∥ TA_RNG1 ∥ TA_RNG2 ∥ TA_RNG3)
```

TA_ASM	0x00000000	対象タスクがアセンブラで書かれている
TA_HLNG	0x00000001	対象タスクが高級言語で書かれている
TA_USERBUF	0x00000020	スタック領域としてユーザが指定した領域を使用する
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する
TA_RNG0	0x00000000	対象タスクは保護レベル0で実行する
TA_RNG1	0x00000100	対象タスクは保護レベル1で実行する
TA_RNG2	0x00000200	対象タスクは保護レベル2で実行する
TA_RNG3	0x00000300	対象タスクは保護レベル3で実行する

タスクの生成情報

TA_HLNGの指定を行った場合には、タスク起動時に直接taskのアドレスにジャンプするのではなく、高級言語の環境設定プログラム(高級言語対応ルーチン)を通してからtaskのアドレスにジャンプします。TA_HLNG属性の場合、タスクは次の形式となります。

```
void task( INT stacd, VP exinf )
{
    /* 処理 */
    tk_ext_tsk( ); または tk_exd_tsk( ); /* タスクの終了 */
}
```

タスクの起動パラメータとして、tk_sta_taskで指定するタスク起動コードstacd、およびtk_cre_taskで指定する拡張情報exinfを渡します。関数からの単純なリターン(return)でタスクを終了することはできません。

TA_ASM属性を指定した場合のタスクの形式は実装依存となります。ただし、起動パラメータとしてstacd、exinfを渡さなければならなりません。

各タスクはスタックを1本持ちます。μT-Kernelには保護レベルの機能が無いため、T-Kernelのようなユーザスタックとシステムスタックの使い分けは存在しません。

TA_USERBUFを指定した場合にbufptrが有効になり、bufptrを先頭とするstkszバイトのメモリ領域をスタック領域として使用します。この場合、スタックはOSで用意しません。TA_USERBUFを指定しなかった場合はbufptrは無視され、スタック領域はOSが確保します。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールのみ操作可能です。

タスクの生成例

```
 1: #include <tk/tkernel.h>
 2: IMPORT void task(INT, VP);
 3: ID tskid;
 4:
 5: EXPORT void create(void)
 6: {
 7: T_CTSK t_ctsk;
 8:     t_ctsk.exinf   = 0;              // 拡張情報の指定は自由
 9:     t_ctsk.tskatr  = TA_HLNG;        // 高級言語、保護レベル０を指定
10:     t_ctsk.task    = task;           // タスクの起動アドレス
11:     t_ctsk.itskpri = 5;              // 初期優先度の指定は自由
12:     t_ctsk.stksz   = 1024;           // スタックサイズの指定
13:     tskid = tk_cre_tsk( &t_ctsk );   // tk_cre_tskシステムコールにより生成
14:     if( tskid > E_OK )               // エラーコードのチェック
15:         tk_sta_tsk( tskid, 0 );      // tk_sta_tskシステムコールで起動
16: }                                    // タスクの起動コードの指定は自由
```

● 生成するタスクの例

```
EXPORT void task(INT stacd, VP exinf)
{
  // タスクに与えられた処理
  tk_ext_tsk( ); // またはtk_exd_tsk( );
}
```

タスクの生成例

3行目：タスクID格納用の変数を外部変数として宣言します。

7行目：タスク生成に必要なパラメータパケット（t_ctsk）を宣言します。

8行目：タスクに渡す拡張情報をexinfに設定します。設定した値はタスクの第２引数に渡されます。

9行目：タスクの属性をtskatrに設定します。生成するタスクはC言語で記述されているためTA_HLNGを指定します。また、保護レベルは何もしていないため、TA_RNG0の保護レベル０を指定したことになります。さらにTA_USERBUFを指定していないため、bufptrは無効な項目となり、タスクのスタック領域はOSが確保します。

10行目：タスクの起動アドレスをtaskに設定します。

11行目：タスクの初期優先度をitskpriに設定します。

12行目：タスクのスタックサイズをstkszに設定します。

13行目：tk_cre_tskシステムコールでタスクを生成し、リターンコードを変数tskidに格納します。

14行目：タスクが正常に生成できたかどうかをif文でチェックします。正の値であれば正常に生成できているため、E_OK（ゼロ）より大きいかを調べます。

15行目：正常に生成できたのであれば、tk_sta_tskシステムコールでタスクを起動します。tk_sta_tskシステムコールの第２引数はタスクの第１引数に渡されます。

タスク管理機能のコーディング例

タスクＡ

```
1: #include <tk/tkernel.h>
2: IMPORT ID tsk_b_id;                  // tsk_bのID番号格納用変数
3:
4: EXPORT void tsk_a(INT stacd, VP exinf)
5: {
6:     tk_sta_tsk( tsk_b_id, 0 );      // tk_sta_tskシステムコールでtsk_bを起動
7: //  タスクに与えられた処理
8:     tk_ext_tsk( );                  // tk_ext_tskシステムコールで終了
9: }
```

タスクＢ

```
1: #include <tk/tkernel.h>
2:
3: EXPORT void tsk_b(INT stacd, VP exinf)
4: {
5: //  タスクに与えられた処理
6:     tk_ext_tsk( );                  // tk_ext_tskシステムコールで終了
7: }
```

タスクＡ

2行目：上記の例ではタスクＢのID番号は変数tsk_b_idに格納されていると仮定しています。
6行目：tk_sta_tskシステムコールを発行してタスクＢを起動します。
7行目：タスクに与えられた処理を実行します。
8行目：tk_ext_tskシステムコールを発行して終了します。

タスクＢ

5行目：タスクに与えられた処理を実行します。
6行目：tk_ext_tskシステムコールを発行して終了します。

3.2　タスク付属同期機能

タスク付属同期機能のシステムコール一覧

■自タスクを起床待ち状態へ移行
```
ER ercd    = tk_slp_tsk( TMO tmout );
```
■他タスクの起床
```
ER ercd    = tk_wup_tsk( ID tskid );
```
■タスクの起床要求を無効化
```
INT wupcnt = tk_can_wup( ID tskid );
```
■他タスクの待ち状態解除
```
ER ercd    = tk_rel_wai( ID tskid );
```
■他タスクを強制待ち状態へ移行
```
ER ercd    = tk_sus_tsk( ID tskid );
```
■強制待ち状態のタスクを再開
```
ER ercd    = tk_rsm_tsk( ID tskid );
```
■強制待ち状態のタスクを強制再開
```
ER ercd    = tk_frsm_tsk( ID tskid );
```
■タスク遅延
```
ER ercd    = tk_dly_tsk( RELTIM dlytim );
```

●tk_wup_tskによる起床要求キューイング数の限界値は実装依存（RX600用μT-Kernelは2,147,483,647回）
●tk_sus_tskによる強制待ち要求のネスト数の限界値も実装依存（RX600用μT-Kernelは2,147,483,647回）
●tk_slp_tsk(TMO_FEVR)がμITRON4.0仕様のslp_tskに対応

タスク付属同期機能は、タスクの状態を直接的に操作することによって同期を行うための機能です。タスクを起床待ちにする機能とそこから起床する機能、タスクの起床要求をキャンセルする機能、タスクの待ち状態を強制解除する機能、タスクを強制待ち状態へ移行する機能とそこから再開する機能、自タスクの実行を遅延する機能が含まれます。

タスクに対する起床要求は、キューイングされます。すなわち、起床待ち状態でないタスクを起床しようとすると、そのタスクを起床しようとしたという記録が残り、後でそのタスクが起床待ちに移行しようとした時に、タスクを起床待ち状態にしません。タスクに対する起床要求のキューイングを実現するために、タスクは起床要求キューイング数を持ちます。タスクの起床要求キューイング数は、タスクの起動時に0にクリアされます。

タスクに対する強制待ち要求は、ネストされます。すなわち、すでに強制待ち状態（二重待ち状態を含む）になっているタスクを再度強制待ち状態に移行させようとすると、そのタスクを強制待ち状態に移行させようとしたという記録が残り、後でそのタスクを強制待ち状態（二重待ち状態を含む）から再開させようとした時に、強制待ちからの再開を行いません。タスクに対する強制待ち要求のネストを実現するために、タスクは強制待ち要求ネスト数を持ちます。タスクの強制待ち要求ネスト数は、タスクの起動時に0にクリアします。

【T-Kernel仕様との差異】

以下のタスクイベントと待ち禁止に関するシステムコールはμT-Kernel仕様には存在しません。

- tk_sig_tev タスクイベントの送信
- tk_wai_tev タスクイベント待ち
- tk_dis_wai タスク待ち状態の禁止
- tk_ena_wai タスク待ち禁止の解除

タスク付属同期機能のコーディング例

タスクＡ

```
1: #include <tk/tkernel.h>
2: IMPORT ID tsk_b_id;                  // tsk_bのID番号格納用変数
3:
4: EXPORT void tsk_a(INT stacd, VP exinf)
5: {
6:     tk_wup_tsk( tsk_b_id );          // tk_wup_tskシステムコールでtsk_bを起床
7: //  タスクに与えられた処理
8:     tk_ext_tsk( );                   // tk_ext_tskシステムコールで終了
9: }
```

タスクＢ

```
1: #include <tk/tkernel.h>
2:
3: EXPORT void tsk_b(INT stacd, VP exinf)
4: {
5:     while( 1 )  {
6:         tk_slp_tsk( TMO_FEVR );    // tk_slp_tskシステムコールで起床待ち
7: //      タスクに与えられた処理
8:     }
9: }
```

タスクＡ

2行目：上記の例ではタスクＢのID番号は変数tsk_b_idに格納されていると仮定しています。
6行目：tk_wup_tskシステムコールを発行してタスクＢを起床します。
7行目：タスクに与えられた処理を実行します。
8行目：tk_ext_tskシステムコールを発行して終了します。

タスクＢ

6行目：TMO_FEVRの永久待ちでtk_slp_tskシステムコールを発行して起床待ちとします。
7行目：タスクに与えられた処理を実行します。

3.3　同期・通信機能

3.3.1　セマフォ

セマフォのシステムコール一覧

■セマフォ生成
```
ID semid = tk_cre_sem( T_CSEM *pk_csem );
```
■セマフォ削除
```
ER ercd  = tk_del_sem( ID semid );
```
■セマフォ資源返却
```
ER ercd  = tk_sig_sem( ID semid, INT cnt );
```
■セマフォ資源獲得
```
ER ercd  = tk_wai_sem( ID semid, INT cnt, TMO tmout );
```
■セマフォ状態参照
```
ER ercd  = tk_ref_sem( ID semid, T_RSEM *pk_rsem );
```

●μT-Kernel仕様におけるセマフォの特徴
- 最大資源数の最低値は32,767個、それ以上は実装依存（RX600用μT-Kernelは2,147,483,647個）
- 資源獲得・返却は一度に複数個が指定可能
- 待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- FIFO順、優先度順の各々に待ち行列先頭のタスクを優先(TA_FIRST)、要求数の少ないタスクを優先(TA_CNT)が選択可能

●tk_sig_sem(***, 1)がμITRON仕様のsig_semに対応

●tk_wai_sem(***, 1, TMO_FEVR)がμITRON4.0仕様のwai_semに対応
tk_wai_sem(***, 1, TMO_POL) がμITRON4.0仕様のpol_semに対応

セマフォは、使用されていない資源の有無や数量を数値で表現することにより、その資源を使用する際の排他制御や同期を行うためのオブジェクトです。セマフォ機能には、セマフォを生成 / 削除する機能、セマフォの資源を獲得/返却する機能、セマフォの状態を参照する機能が含まれます。セマフォはID番号で識別されるオブジェクトであり、セマフォのID番号をセマフォ IDと呼びます。

セマフォは、対応する資源の有無や数量を表現する資源数と、資源の獲得を待つタスクの待ち行列を持ちます。資源を m 個返却する側（イベントを知らせる側）では、セマフォの資源数を m 個増やします。一方、資源を n 個獲得する側（イベントを待つ側）では、セマフォの資源数を n 個減らします。セマフォの資源数が足りなくなった場合（具体的には、資源数を減らすと資源数が負になる場合）、資源を獲得しようとしたタスクは、次に資源が返却されるまでセマフォ資源の獲得待ち状態となります。セマフォ資源の獲得待ち状態になったタスクは、そのセマフォの待ち行列につながれます。

また、セマフォに対して資源が返却され過ぎるのを防ぐために、セマフォ毎に最大資源数を設定することができます。最大資源数を越える資源がセマフォに返却されようとした場合（具体的には、セマフォの資源数を増やすと最大資源数を越える場合）には、エラーを報告します。

セマフォの生成情報

pk_csemの内容

```
typedef struct t_csem {
    VP    exinf;       /* Extended information          拡張情報            */
    ATR   sematr;      /* Semaphore attribute           セマフォ属性        */
    INT   isemcnt;     /* Semaphore initial count value セマフォの初期値    */
    INT   maxsem;      /* Semaphore maximum count value セマフォの最大値    */
    UB    dsname[8];   /* Object name                   DSオブジェクト名称 */
} T_CSEM
```

```
sematr := (TA_TFIFO // TA_TPRI) | (TA_FIRST // TA_CNT) | [TA_DSNAME]

TA_TFIFO     0x00000000    待ちタスクのキューイングはFIFO
TA_TPRI      0x00000001    待ちタスクのキューイングは優先度順
TA_FIRST     0x00000000    待ち行列先頭のタスクを優先
TA_CNT       0x00000002    要求数の少ないタスクを優先
TA_DSNAME    0x00000040    DSオブジェクト名称を指定する
```

セマフォの生成情報

TA_TFIFO、TA_TPRI では、タスクがセマフォの待ち行列に並ぶ際の並び方を指定することができます。属性がTA_TFIFOであればタスクの待ち行列はFIFOとなり、属性がTA_TPRIであればタスクの待ち行列はタスクの優先度順とります。

TA_FIRST、TA_CNTでは、資源獲得の優先順を指定します。TA_FIRSTおよびTA_CNTの指定によって待ち行列の並び順が変わることはありません。待ち行列の並び順はTA_TFIFO、TA_TPRIによってのみ決定されます。

TA_FIRSTでは、要求カウントに関係なく待ち行列の先頭のタスクから順に資源を割当てます。待ち行列の先頭のタスクが要求分の資源を獲得できない限り、待ち行列の後ろのタスクが資源を獲得することはありません。

TA_CNTでは、要求カウント分の資源が獲得できるタスクから順に割当てます。具体的には、待ち行列の先頭のタスクから順に要求カウント数を検査し、要求カウント数分の資源が割当てられるタスクに割当てます。要求カウント数の少ない順に割当てる訳ではありません。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

セマフォの生成例

```
 1: #include <tk/tkernel.h>
 2: ID semid;                                    // セマフォID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CSEM t_csem;
 7:     t_csem.sematr  = TA_TFIFO | TA_FIRST;    // FIFO順、待ち行列先頭を優先
 8:     t_csem.isemcnt = 1;                      // セマフォの初期値は1
 9:     t_csem.maxsem  = 1;                      // セマフォの最大値も1
10:     semid = tk_cre_sem( &t_csem );           // tk_cre_semにより生成
11:     if( semid > E_OK )                       // エラーコードのチェック
12:       return TRUE;                           // 正常終了
13:     else
14:       return FALSE;                          // 異常終了
15: }
```

セマフォの生成例

2行目： セマフォ ID格納用の変数を外部変数として宣言します。

6行目： セマフォ生成に必要なパラメータパケット（t_csem）を宣言します。

7～9行目： パラメータパケットに生成するセマフォの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： セマフォの属性をsematrに設定します。待ちタスクのキューイングはFIFO、待ち行列先頭のタスクを優先とするためTA_TFIFOとTA_FIRSTを指定します。

8行目： セマフォの初期値をisemcntに設定します。

9行目： セマフォの最大値をmaxsemに設定します。

10行目： tk_cre_semシステムコールでセマフォを生成し、リターンコードを変数semidに格納します。

11～14行目： セマフォが正常に生成できたかどうかをif文でチェックします。正の値であれば正常に生成できているためTRUEを返し、そうでなければFALSEを返します。

セマフォのコーディング例

タスクＡ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID semid;                          // セマフォID番号格納用変数
 3:
 4: EXPORT void tsk_a(INT stacd, VP exinf)
 5: {
 6:     tk_wai_sem( semid, 1, TMO_FEVR );     // 資源の獲得（tk_wai_sem）
 7: //  排他処理
 8:     tk_sig_sem( semid, 1 );               // 資源の返却（tk_sig_sem）
 9:     tk_ext_tsk( );
10: }
```

タスクＢ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID semid;                          // セマフォID番号格納用変数
 3:
 4: EXPORT void tsk_b(INT stacd, VP exinf)
 5: {
 6:     tk_wai_sem( semid, 1, TMO_FEVR );     // 資源の獲得（tk_wai_sem）
 7: //  排他処理
 8:     tk_sig_sem( semid, 1 );               // 資源の返却（tk_sig_sem）
 9:     tk_ext_tsk( );
10: }
```

タスクＡ、タスクＢ

上記の例は資源数、初期値が共に１のセマフォを使用して排他制御を行うものです。

2行目：上記の例では排他制御用セマフォのID番号は変数semidに格納されていると仮定しています。
6行目：獲得資源数は１、TMO_FEVRの永久待ちでtk_wai_semシステムコールを発行し、資源の獲得を行います。
7行目：排他処理を行います。
8行目：返却資源数は１でtk_sig_semシステムコールを発行し、資源の返却を行います。

3.3.2　イベントフラグ
イベントフラグのサービルコール一覧

■イベントフラグ生成
```
ID flgid = tk_cre_flg( T_CFLG *pk_cflg );
```
■イベントフラグ削除
```
ER ercd  = tk_del_flg( ID flgid );
```
■イベントフラグのセット
```
ER ercd  = tk_set_flg( ID flgid, UINT setptn );
```
■イベントフラグのクリア
```
ER ercd  = tk_clr_flg( ID flgid, UINT clrptn );
```
■イベントフラグ待ち
```
ER ercd  = tk_wai_flg( ID flgid, UINT waiptn, UINT wfmode, UINT *p_flgptn, TMO tmout );
```
■イベントフラグ状態参照
```
ER ercd  = tk_ref_flg( ID flgid, T_RSEM *pk_rflg );
```

●μT-Kernel仕様におけるイベントフラグの特徴
- 1つのイベントフラグで扱うビット数は対応プロセッサの1ワード分（RX600用μT-Kernelは32ビット）
- 待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- 複数タスクの待ちを許さない(TA_WSGL)、許す(TA_WMUL) が選択可能
- 待ちモードでは、AND待ち(TWF_ANDW)、OR待ち(TWF_ORW)、全クリア指定(TWF_CLR)、条件ビットのみクリア指定(TWF_BITCLR)が選択可能

●tk_wai_flg(***, ***, ***, ***, TMO_FEVR)がμITRON4.0仕様のwai_flgに対応
tk_wai_flg(***, ***, ***, ***, TMO_POL) がμITRON4.0仕様のpol_flgに対応

イベントフラグは、イベントの有無をビット毎のフラグで表現することにより、同期を行うためのオブジェクトです。イベントフラグ機能には、イベントフラグを生成/削除する機能、イベントフラグをセット/クリアする機能、イベントフラグで待つ機能、イベントフラグの状態を参照する機能が含まれます。イベントフラグはID番号で識別されるオブジェクトであり、イベントフラグのID番号をイベントフラグIDと呼びます。

イベントフラグは、対応するイベントの有無をビット毎に表現するビットパターンと、そのイベントフラグで待つタスクの待ち行列を持ちます。イベントフラグのビットパターンを、単にイベントフラグと呼ぶ場合もあります。イベントを知らせる側では、イベントフラグのビットパターンの指定したビットをセットないしはクリアすることが可能です。一方、イベントを待つ側では、イベントフラグのビットパターンの指定したビットのすべてまたはいずれかがセットされるまで、タスクをイベントフラグ待ち状態にすることができます。イベントフラグ待ち状態になったタスクは、そのイベントフラグの待ち行列につながれます。

イベントフラグの生成情報

pk_cflgの内容

```
typedef struct t_cflg {
    VP      exinf;      /* Extended information     拡張情報              */
    ATR     flgatr;     /* Event flag attribute     イベントフラグ属性    */
    UINT    iflgptn;    /* Event flag initial value イベントフラグの初期値 */
    UB      dsname[8];  /* Object name              DSオブジェクト名称    */
} T_CFLG;
```

```
flgatr := (TA_TFIFO ∥ TA_TPRI) | (TA_WMUL ∥ TA_WSGL) | [TA_DSNAME]

TA_TFIFO     0x00000000    待ちタスクのキューイングはFIFO
TA_TPRI      0x00000001    待ちタスクのキューイングは優先度順
TA_WSGL      0x00000000    複数タスクの待ちを許さない（Wait Single Task）
TA_WMUL      0x00000008    複数タスクの待ちを許す（Wait Multiple Task）
TA_DSNAME    0x00000040    DS オブジェクト名称を指定する
```

イベントフラグの生成情報

TA_WSGLを指定した場合は、複数のタスクが同時に待ち状態になることを禁止します。TA_WMULを指定した場合は、同時に複数のタスクが待ち状態となることが許されます。

TA_TFIFO、TA_TPRIでは、タスクがイベントフラグの待ち行列に並ぶ際の並び方を指定することができます。属性がTA_TFIFOであればタスクの待ち行列はFIFOとなり、属性がTA_TPRIであればタスクの待ち行列はタスクの優先度順となります。ただし、TA_WSGLを指定した場合は待ち行列を作らないため、TA_TFIFO、TA_TPRIのどちらを指定しても動作に変わりはありません。

複数のタスクが待っている場合、待ち行列の先頭から順に待ち条件が成立しているか検査し、待ち条件が成立しているタスクの待ちを解除する。したがって、待ち行列の先頭のタスクが必ずしも最初に待ちが解除される訳ではありません。また、待ち条件が成立したタスクが複数あれば、複数のタスクの待ちが解除されます。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

イベントフラグの生成例

```
 1: #include <tk/tkernel.h>
 2: ID flgid;                                    // イベントフラグID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CFLG t_cflg;
 7:     t_cflg.flgatr  = TA_WSGL;                // 複数タスクの待ちを許さない
 8:     t_cflg.iflgptn = 0;                      // イベントフラグの初期値は0
 9:     flgid = tk_cre_flg( &t_cflg );           // tk_cre_flgにより生成
10:     return flgid > E_OK ? TRUE : FALSE;      // エラーコードのチェック
11: }
```

イベントフラグの生成例

2行目： イベントフラグID格納用の変数を外部変数として宣言します。

6行目： イベントフラグ生成に必要なパラメータパケット（t_cflg）を宣言します。

7～8行目： パラメータパケットに生成するイベントフラグの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： イベントフラグの属性をflgatrに設定します。複数タスクの待ちを許さないためTA_WSGLを指定します。この場合、待ち行列を作らないため、TA_TFIFO、TA_TPRIはどちらも指定しません（どちらを指定しても構いません）。

8行目： イベントフラグの初期ビットパターンであるiflgptnにはゼロを指定します。この場合、生成時に設定されているイベントが存在しないことを意味します。

9行目： tk_cre_flgシステムコールでイベントフラグを生成し、リターンコードを変数flgidに格納します。

10行目： イベントフラグが正常に生成できたかどうかを条件式でチェックします。正の値であれば正常に生成できているためTRUEを返し、そうでなければFALSEを返します。

イベントフラグのコーディング例（タスク同期）

タスクＡ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID flgid;                          // イベントフラグID番号格納用変数
 3:
 4: EXPORT void tsk_a(INT stacd, VP exinf)
 5: {
 6: //  タスクに与えられた処理
 7:     tk_set_flg( flgid, 0x01 );            // tk_set_flgによるイベントの設定
 8:     tk_ext_tsk( );
 9: }
```

タスクＢ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID flgid;                          // イベントフラグID番号格納用変数
 3:
 4: EXPORT void tsk_b(INT stacd, VP exinf)
 5: {
 6: //  タスクに与えられた処理
 7:     tk_set_flg( flgid, 0x20 );            // tk_set_flgによるイベントの設定
 8:     tk_ext_tsk( );
 9: }
```

タスクＣ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID flgid;                          // イベントフラグID番号格納用変数
 3:
 4: EXPORT void tsk_c(INT stacd, VP exinf)
 5: {
 6: UNIT flgptn;
 7:     while( 1 )  {                        // tk_wai_flgによるイベントの待ち
 8:         tk_wai_flg( flgid, 0x21, TWF_ANDW | TWF_CLR, &flgptn, TMO_FEVR );
 9:         //  タスクに与えられた処理
10:     }
11: }
```

タスクＡ、タスクＢ

2行目：上記の例ではイベントフラグのID番号は変数flgidに格納されていると仮定しています。
6行目：タスクに与えられた処理を実行します。
7行目：tk_set_flgシステムコールを発行して処理終了のイベントを設定します。

タスクＣ

2行目：上記の例ではイベントフラグのID番号は変数flgidに格納されていると仮定しています。
8行目：tk_wai_flgシステムコールを発行してタスクＡとタスクＢの処理終了のイベントを待ちます。
9行目：タスクに与えられた処理を実行します。

イベントフラグのコーディング例（データ受け渡し）

割り込みハンドラ

```
 1: #include <tk/tkernel.h>
 2: #include "iodefine.h"
 3: IMPORT ID flgid;                          // イベントフラグID番号格納用変数
 4:
 5: EXPORT void void sci_hdr(UINT dintno)
 6: {
 7:     tk_set_flg( flgid, SCI0.RDR );        // 受信データをtk_set_flgで設定
 8: }
```

タスク

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID flgid;                          // イベントフラグID番号格納用変数
 3:
 4: EXPORT void task(INT stacd, VP exinf)
 5: {
 6: UINT flgptn;
 7:     while( 1 )  {                        // tk_wai_flgで受信データを読込
 8:         tk_wai_flg( flgid, 0xFF, TWF_ORW | TWF_CLR, &flgptn, TMO_FEVR );
 9:         //  タスクに与えられた処理
10:     }
11: }
```

割り込みハンドラ

2行目：上記の例ではイベントフラグのID番号は変数flgidに格納されていると仮定しています。
7行目：tk_set_flgシステムコールを発行してSCIの受信データをイベントフラグに設定します。

タスク

2行目：上記の例ではイベントフラグのID番号は変数flgidに格納されていると仮定しています。
8行目：tk_wai_flgシステムコールを発行し、受信データがイベントとして設定されるのを待ちます。
9行目：システムコール完了後は、リターンパラメータとして変数flgptnに設定された受信データを使用し、タスクに与えられた処理を行います。

3.3.3　メールボックス

メールボックスのシステムコール一覧

■メールボックス生成
```
ID mbxid = tk_cre_mbx( T_CMBX *pk_cmbx );
```
■メールボックス削除
```
ER ercd  = tk_del_mbx( ID mbxid );
```
■メールボックスへ送信
```
ER ercd  = tk_snd_mbx( ID mbxid, T_MSG *pk_msg );
```
■メールボックスから受信
```
ER ercd  = tk_rcv_mbx( ID mbxid, T_MSG **ppk_msg, TMO tmout );
```
■メールボックス状態参照
```
ER ercd  = tk_ref_mbx( ID mbxid, T_RMBX *pk_rmbx );
```

●μT-Kernel仕様におけるメールボックスの特徴
- 待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- メッセージの待ち行列はFIFO順(TA_MFIFO)、優先度順(TA_MPRI) が選択可能

●tk_rcv_mbx(***, ***, TMO_FEVR)がμITRON4.0仕様の rcv_mbxに対応
tk_rcv_mbx(***, ***, TMO_POL) がμITRON4.0仕様のprcv_mbxに対応

メールボックスは、メモリ上に置かれたメッセージを受渡しすることにより、同期と通信を行うためのオブジェクトです。メールボックス機能には、メールボックスを生成/削除する機能、メールボックスに対してメッセージを送信/受信する機能、メールボックスの状態を参照する機能が含まれます。メールボックスはID番号で識別されるオブジェクトであり、メールボックスのID番号をメールボックスIDと呼びます。

メールボックスは、送信されたメッセージを入れるためのメッセージキューと、メッセージの受信を待つタスクの待ち行列を持ちます。メッセージを送信する側(イベントを知らせる側)では、送信したいメッセージをメッセージキューに入れます。一方、メッセージを受信する側(イベントを待つ側)では、メッセージキューに入っているメッセージを一つ取り出します。メッセージキューにメッセージが入っていない場合は、次にメッセージが送られてくるまでメールボックスからの受信待ち状態になります。メールボックスからの受信待ち状態になったタスクは、そのメールボックスの待ち行列につながれます。

【T-Kernel仕様との差異】

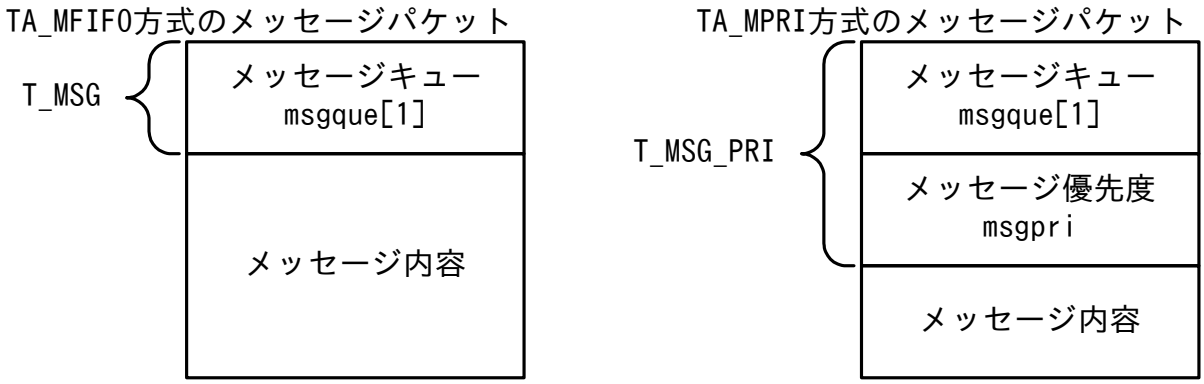
μT-Kernel仕様にはタスク固有空間の概念が無く、T-Kernel仕様でいうところの共有空間しか存在しない。メールボックスで実際に送受信されるのはメモリ上に置かれたメッセージの先頭のポインタであるため、T-Kernel仕様で使用する場合はメッセージがタスク固有空間ではなく、共有空間に置かれるように注意する必要がある。μT-Kernel仕様では全てが共有空間になるのでこのことを意識する必要はない。

メッセージパケット

メッセージヘッダの宣言

```
/* Mail box message header */
typedef struct t_msg {
        VP      msgque[1];      /* Area for message queue */
} T_MSG;

typedef struct t_msg_pri {
        T_MSG   msgque;         /* Area for message queue */
        PRI     msgpri;         /* Message priority       */
} T_MSG_PRI;
```



メールボックスによって実際に送受信されるのは、メモリ上に置かれたメッセージの先頭番地のみです。すなわち、送受信されるメッセージの内容のコピーは行われません。カーネルは、メッセージキューに入っているメッセージをリンクリストにより管理しています。このため、アプリケーションプログラムは、送信するメッセージの先頭にカーネルがリンクリストに用いるための領域を確保しなければなりません。この領域をメッセージヘッダと呼びます。また、メッセージヘッダと、それに続くアプリケーションがメッセージを入れるための領域をあわせて、メッセージパケットと呼びます。

メールボックスへメッセージを送信するシステムコールは、メッセージパケットの先頭番地（pk_msg）をパラメータとして渡します。また、メールボックスからメッセージを受信するシステムコールは、メッセージパケットの先頭番地をリターンパラメータとして受け取ります。

メッセージキューをメッセージの優先度順にする場合には、メッセージの優先度を入れるための領域(msgpri)も、メッセージヘッダ中に持つ必要があります。

ユーザが実際にメッセージを入れることができるのは、メッセージ先頭アドレスの直後からではなく、メッセージヘッダの後の部分からです。

カーネルは、メッセージキューに入っている（ないしは、入れようとしている）メッセージのメッセージヘッダ（メッセージ優先度のための領域を除く）の内容を書き換えます。一方、アプリケーションは、メッセージキューに入っているメッセージのメッセージヘッダ(メッセージ優先度のための領域を含む)の内容を書き換えてはなりません。

メールボックスの生成情報

pk_cmbxの内容

```
typedef struct t_cmbx {
    VP    exinf;       /* Extended information 拡張情報          */
    ATR   mbxatr;      /* Mail box attribute   メールボックス属性 */
    UB    dsname[8];   /* Object name          DSオブジェクト名称 */
} T_CMBX;
```

mbxatr := (TA_TFIFO ∥ TA_TPRI) | (TA_MFIFO ∥ TA_MPRI) | [TA_DSNAME]

TA_TFIFO	0x00000000	待ちタスクのキューイングはFIFO
TA_TPRI	0x00000001	待ちタスクのキューイングは優先度順
TA_MFIFO	0x00000000	メッセージのキューイングはFIFO
TA_MPRI	0x00000002	メッセージのキューイングは優先度順
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する

メールボックスの生成情報

TA_TFIFO、TA_TPRI では、メッセージを受信するタスクがメールボックスの待ち行列に並ぶ際の並び方を指定することができます。属性がTA_TFIFOであればタスクの待ち行列はFIFOとなり、属性がTA_TPRIであればタスクの待ち行列はタスクの優先度順となります。

一方、TA_MFIFO、TA_MPRIでは、メッセージがメッセージキュー（受信されるのを待つメッセージの待ち行列）に入る際の並び方を指定することができます。属性がTA_MFIFOであればメッセージキューはFIFOとなり、属性がTA_MPRI であればメッセージキューはメッセージの優先度順となります。メッセージの優先度は、メッセージパケットの中の特定領域で指定します。メッセージ優先度は正の値であり、1 が最も優先度が高く、数値が大きくなるほど優先度は低くなり、PRI 型で表わせる最大の正の値が最も低い優先度となります。また、同一優先度の場合はFIFOとなります。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

メールボックスの生成例

```
 1: #include <tk/tkernel.h>
 2: ID mbxid;                                    // メールボックスID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CMBX t_cmbx;
 7:     t_cmbx.mbxatr  = TA_TFIFO | TA_MFIFO;    // 待ちタスクキューはFIFO
 8:                                              // メッセージキューもFIFO
 9:     mbxid = tk_cre_mbx( &t_cmbx );           // tk_cre_mbxにより生成
10:     return mbxid > E_OK;                     // エラーコードのチェック
11: }
```

メールボックスの生成例

2行目： メールボックスID格納用の変数を外部変数として宣言します。

6行目： メールボックス生成に必要なパラメータパケット（t_cmbx）を宣言します。

7～8行目： パラメータパケットに生成するメールボックスの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： メールボックスの属性をmtxatrに設定します。待ちタスクとメッセージのキューイングを共にFIFOとするため、TA_TFIFOとTA_MFIFOを指定します。

9行目： tk_cre_mbxシステムコールでメールボックスを生成し、リターンコードを変数mbxidに格納します。

10行目： メールボックスが正常に生成できたかどうかを条件式でチェックします。比較演算子の結果は「真」または「偽」であるため、その値をそのまま返します。

メッセージパケットのコーディング例

ユーザ固有のメッセージパケット（user.h）

```
1: typedef struct user_msg {
2:     T_MSG    msghead;            // メッセージキュー（OS管理エリア）
3:     ID       tskid;              // 要求元タスクID
4:     char    *data;               // 送信データの先頭アドレス
5: } USER_MSG;
```

メッセージパケットの制約事項
【TA_MFIFOの場合】
メッセージパケットの先頭にはT_MSG型のメンバを設けなければならない。
【TA_MPRIの場合】
メッセージパケットの先頭にはT_MSG_PRI型のメンバを設けなければならない。

上記以外のメンバ構成はユーザ側で自由に定義して構わない。

メッセージパケット（user.h）

1～5行目： ユーザ固有のメッセージパケットの型宣言です。
2行目： メッセージのキューイングがTA_MFIFOの場合、T_MSG型のmsgheadメンバを設けます。
3行目： メッセージ送信元のタスクIDを格納するtskidメンバを設けます。
4行目： 送信データのアドレスを設定するdataメンバを設けます。
5行目： 上記のメッセージパケットをUSER_MSG型として宣言します。

メールボックスのコーディング例

タスクA

```
 1: #include <tk/tkernel.h>
 2: #include "user.h"                          // ユーザ固有ヘッダ
 3: IMPORT ID mbxid;                           // メールボックスID番号格納用変数
 4: LOCAL USER_MSG msg;                        // メッセージパケットの本体
 5:
 6: EXPORT void tsk_a(INT stacd, VP exinf)
 7: {
 8:     msg.tskid = tk_get_tid( );             // 自タスクのIDをtk_get_tidで設定
 9:     msg.data = "RENESAS";                  // 送信データを設定
10:     tk_snd_mbx( mbxid, (T_MSG *)&msg );   // tk_snd_mbxでメッセージを送信
11:     tk_slp_tsk( TMO_FEVR );               // 送信メッセージの処理待ち
12:     tk_ext_tsk( );
13: }
```

タスクB

```
 1: #include <tk/tkernel.h>
 2: #include "user.h"                          // ユーザ固有ヘッダ
 3: IMPORT ID mbxid;                           // メールボックスID番号格納用変数
 4:
 5: EXPORT void tsk_b(INT stacd, VP exinf)
 6: {
 7: USER_MSG *msg;                             // メッセージパケットのポインタ
 8:     while( 1 )  {                          // tk_rcv_mbxでメッセージを受信
 9:         tk_rcv_mbx( mbxid, (T_MSG **)&msg, TMO_FEVR );
10:         //  受信したメッセージに対する処理
11:         tk_wup_tsk( msg->tskid );          // 処理待ちのタスクを起床
12:     }
13: }
```

タスクA

3行目： 上記の例ではメールボックスのID番号は変数mbxidに格納されていると仮定しています。
4行目： メッセージパケットの本体をmsg変数として宣言します。
8～9行目： メッセージパケットに対して必要な情報を設定します。tskidメンバには自タスクのIDを設定するため、tk_get_tidシステムコールでID番号を取得します。
10行目： tk_snd_mbxシステムコールでメッセージパケットを送信します。
11行目： tk_slp_tskシステムコールで送信したメッセージの処理終了を待ちます。

タスクB

3行目： 上記の例ではメールボックスのID番号は変数mbxidに格納されていると仮定しています。
7行目： メッセージパケットを指すmsgポインタ変数を宣言します。
9行目： tk_rcv_mbxシステムコールを発行し、メッセージの受信を待ちます。受信したメッセージの先頭アドレスはmsg変数に格納します。
10行目： 受信したメッセージに対する処理を実行します。
11行目： 受信したメッセージに対する処理の終了後は要求元のタスクをtk_slp_tskシステムコールで起床します。

3.4　拡張同期・通信機能

3.4.1　ミューテックス

ミューテックスのシステムコール一覧

■ミューテックス生成
```
ID mtxid = tk_cre_mtx( T_CMTX *pk_cmtx );
```
■ミューテックス削除
```
ER ercd  = tk_del_mtx( ID mtxid );
```
■ミューテックスのロック
```
ER ercd  = tk_loc_mtx( ID mtxid, TMO tmout );
```
■ミューテックスのアンロック
```
ER ercd  = tk_unl_mtx( ID mtxid );
```
■ミューテックス状態参照
```
ER ercd  = tk_ref_mtx( ID mtxid, T_RMTX *pk_rmtx );
```

●μT-Kernel仕様におけるミューテックスの特徴
- 待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- 優先度順は優先度順(TA_TPRI)、優先度継承プロトコル(TA_INHERIT)、優先度上限プロトコル(TA_CEILING)が選択可能
- プロトコルは優先度継承プロトコル(TA_INHERIT)、優先度上限プロトコル(TA_CEILING) が指定可能

●tk_loc_mtx(***, TMO_FEVR)がμITRON4.0仕様の loc_mtxに対応
tk_loc_mtx(***, TMO_POL) がμITRON4.0仕様のploc_mtxに対応

ミューテックスは、共有資源を使用する際にタスク間で排他制御を行うためのオブジェクトです。ミューテックスは、排他制御に伴う上限のない優先度逆転を防ぐための機構として、優先度継承プロトコル（priority inheritance protocol)と優先度上限プロトコル(priority ceiling protocol)をサポートします。

ミューテックス機能には、ミューテックスを生成/削除する機能、ミューテックスをロック/ロック解除する機能、ミューテックスの状態を参照する機能が含まれます。ミューテックスはID番号で識別されるオブジェクトであり、ミューテックスのID番号をミューテックスIDと呼びます。

ミューテックスは、ロックされているかどうかの状態と、ロックを待つタスクの待ち行列を持ちます。また、カーネルは、各ミューテックスに対してそれをロックしているタスクを、各タスクに対してそれがロックしているミューテックスの集合を管理します。タスクは、資源を使用する前に、ミューテックスをロックします。ミューテックスが他のタスクにロックされていた場合には、ミューテックスがロック解除されるまで、ミューテックスのロック待ち状態となります。ミューテックスのロック待ち状態になったタスクは、そのミューテックスの待ち行列につながれます。タスクは、資源の使用を終えると、ミューテックスのロックを解除します。

ミューテックスは、ミューテックス属性にTA_INHERIT(=0x02)を指定することにより優先度継承プロトコルを、TA_CEILING(=0x03)を指定することにより優先度上限プロトコルをサポートします。TA_CEILING属性のミューテックスに対しては、そのミューテックスをロックする可能性のあるタスクの中で最も高いベース優先度を持つタスクのベース優先度を、ミューテックス生成時に上限優先度として設定します。TA_CEILING属性のミューテックスを、その上限優先度よりも高いベース優先度を持つタスクがロックしようとした場合、E_ILUSE エラーとなります。また、TA_CEILING 属性のミューテックスをロックしているかロックを待っているタスクのベース優先度を、tk_chg_pri によってそのミューテックスの上限優先度よりも高く設定しようとした場合、tk_chg_pri が E_ILUSE エラーを返します。

これらのプロトコルを用いた場合、上限の無い優先度逆転を防ぐために、ミューテックスの操作に伴ってタスクの現在優先度を変更する。タスクの現在優先度は、次に挙げる優先度の最高値に常に一致するように変更する必要があります。

- タスクのベース優先度
- タスクがTA_INHERIT属性のミューテックスをロックしている場合、それらのミューテックスのロックを待っているタスクの中で最も高い現在優先度を持つタスクの現在優先度
- タスクがTA_CEILING属性のミューテックスをロックしている場合、それらのミューテックス中で最も高い上限優先度を持つミューテックスの上限優先度

ここで TA_INHERIT 属性のミューテックスを待っているタスクの現在優先度が、ミューテックス操作か tk_chg_pri によるベース優先度の変更に伴って変更された場合、そのミューテックスをロックしているタスクの現在優先度の変更が必要になる場合があります。これを推移的な優先度継承と呼びます。さらにそのタスクが、別のTA_INHERIT属性のミューテックスを待っていた場合には、そのミューテックスをロックしているタスクに対して推移的な優先度継承の処理が必要になる場合があります。

ミューテックスの操作に伴ってタスクの現在優先度を変更した場合には、次の処理を行います。

優先度を変更されたタスクが実行できる状態である場合、タスクの優先順位を、変更後の優先度にしたがって変化さます。変更後の優先度と同じ優先度を持つタスクの間では最低の優先順位になるものとします。優先度が変更されたタスクが何らかのタスク優先度順の待ち行列につながれている場合にも、その待ち行列の中での順序を、変更後の優先度にしたがって変化さます。変更後の優先度と同じ優先度を持つタスクの間では最低の順序になるものとします。タスクが終了する時に、そのタスクがロックしているミューテックスが残っている場合には、それらのミューテックスをすべてロック解除します。ロックしているミューテックスが複数ある場合には、それらは後に確保したものから順に解除さます。ロック解除の具体的な処理内容については、tk_unl_mtxの機能説明を参照してください。

【補足事項】

TA_TFIFO属性またはTA_TPRI属性のミューテックスは、最大資源数が1のセマフォ（バイナリセマフォ）と同等の機能を持ちます。ただしミューテックスは、ロックしたタスク以外はロック解除できない、タスク終了時に自動的にロック解除される、などの違いがあります。

ここでいう優先度上限プロトコルは、広い意味での優先度上限プロトコルで、最初に優先度上限プロトコルとして提案されたアルゴリズムではありません。厳密には highest locker protocol などと呼ばれているアルゴリズムです。

ミューテックスの操作に伴ってタスクの現在優先度を変更した結果、優先度を変更されたタスクのタスク優先度順の待ち行列の中での順序が変化した場合、優先度を変更されたタスクないしはその待ち行列で待っている他のタスクの待ち解除が必要になる場合があります。

ミューテックスの生成情報

pk_cmtxの内容

```
typedef struct t_cmtx {
    VP    exinf;       /* Extended information 拡張情報                  */
    ATR   mtxatr;      /* Mutex attribute      ミューテックス属性        */
    PRI   ceilpri;     /* Upper limit priority ミューテックスの上限優先度 */
    UB    dsname[8];   /* Object name          DSオブジェクト名称        */
} T_CMTX;
```

mtxatr := (TA_TFIFO // TA_TPRI // TA_INHERIT // TA_CEILING) | [TA_DSNAME]

TA_TFIFO	0x00000000	待ちタスクのキューイングは FIFO
TA_TPRI	0x00000001	待ちタスクのキューイングは優先度順
TA_INHERIT	0x00000002	優先度継承プロトコル
TA_CEILING	0x00000003	優先度上限プロトコル
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する

ミューテックスの生成情報

TA_TFIFOの場合、ミューテックスのタスクの待ち行列はFIFOとなります。TA_TPRI、TA_INHERIT、TA_CEILINGでは、タスクの優先度順とります。TA_INHERITでは優先度継承プロトコル、TA_CEILINGでは優先度上限プロトコルが適用されます。

TA_CEILINGの場合のみceilpriが有効となり、ミューテックスの上限優先度を設定します。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能である。

ミューテックスの生成例

```
 1: #include <tk/tkernel.h>
 2: ID mtxid;                                    // ミューテックスID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CMTX t_cmtx;
 7:     t_cmtx.mtxatr  = TA_INHERIT;             // 優先度継承プロトコル
 8: //  t_cmtx.mtxatr  = TA_CEILING;             // 優先度上限プロトコル
 9: //  t_cmtx.ceilpri = 5;                      // 上限優先度の指定
10:     mtxid = tk_cre_mtx( &t_cmtx );           // tk_cre_mtxにより生成
11:     return mtxid > E_OK;                     // エラーコードのチェック
12: }
```

ミューテックスの生成例

2行目： ミューテックスID格納用の変数を外部変数として宣言します。

6行目： ミューテックス生成に必要なパラメータパケット（t_cmtx）を宣言します。

7～9行目： パラメータパケットに生成するミューテックスの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： ミューテックスの属性をmtxatrに設定します。TA_INHERITの優先度継承プロトコルやTA_TFIFO、TA_TPRIの待ちタスクのキューイングがFIFOや優先度順の場合、その他に設定する項目はありません。

8～9行目： ミューテックスの属性であるmtxatrに対して、TA_CEILINGTAの優先度上限プロトコルを指定した場合、ceilpriの上限優先度にミューテックスをロックするタスクの中で最も高い優先度を指定します。

10行目： tk_cre_mtxシステムコールでミューテックスを生成し、リターンコードを変数mtxidに格納します。

11行目： ミューテックスが正常に生成できたかどうかを比較演算子で調べ、その結果を返します。

ミューテックスのコーディング例

タスクＡ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID mtxid;                          // ミューテックスID番号格納用変数
 3:
 4: EXPORT void tsk_a(INT stacd, VP exinf)
 5: {
 6:     tk_loc_mtx( mtxid, TMO_FEVR );       // 資源の占有（tk_loc_mtx）
 7: //  排他処理
 8:     tk_unl_mtx( mtxid );                 // 資源の解放（tk_unl_mtx）
 9:     tk_ext_tsk( );
10: }
```

タスクＢ

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID mtxid;                          // ミューテックスID番号格納用変数
 3:
 4: EXPORT void tsk_b(INT stacd, VP exinf)
 5: {
 6:     tk_loc_mtx( mtxid, TMO_FEVR );       // 資源の占有（tk_loc_mtx）
 7: //  排他処理
 8:     tk_unl_mtx( mtxid );                 // 資源の解放（tk_unl_mtx）
 9:     tk_ext_tsk( );
10: }
```

タスクＡ、タスクＢ

2行目：上記の例では排他制御用ミューテックスのID番号は変数mtxidに格納されていると仮定しています。
6行目：TMO_FEVRの永久待ちでtk_loc_mtxシステムコールを発行し、資源の占有を行います。
7行目：排他処理を行います。
8行目：tk_unl_mtxシステムコールを発行し、資源の解放を行います。

3.4.2　メッセージバッファ

メッセージバッファのシステムコール一覧

■メッセージバッファ生成
```
ID mbfid  = tk_cre_mbf( T_CMBF *pk_cmbf );
```
■メッセージバッファ削除
```
ER ercd   = tk_del_mbf( ID mbfid );
```
■メッセージバッファへ送信
```
ER ercd   = tk_snd_mbf( ID mbfid, VP msg, INT msgsz, TMO tmout );
```
■メッセージバッファから受信
```
INT msgsz = tk_rcv_mbf( ID mbfid, VP msg, TMO tmout );
```
■メッセージバッファ状態参照
```
ER ercd   = tk_ref_mbf( ID mbfid, T_RMBF *pk_rmbf );
```

●μT-Kernel仕様におけるメッセージバッファの特徴
- 送信待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- 受信待ちタスクキューの並び方はFIFO順のみ
- メッセージキューの順序はFIFO順のみ

●tk_snd_mbf(***, ***, ***, TMO_FEVR)がμITRON4.0仕様の snd_mbfに対応
tk_snd_mbf(***, ***, ***, TMO_POL) がμITRON4.0仕様のpsnd_mbfに対応

●tk_rcv_mbf(***, ***, TMO_FEVR)がμITRON4.0仕様の rcv_mbfに対応
tk_rcv_mbf(***, ***, TMO_POL) がμITRON4.0仕様のprcv_mbfに対応

メッセージバッファは、可変長のメッセージを受渡しすることにより、同期と通信を行うためのオブジェクトです。メッセージバッファ機能には、メッセージバッファを生成/削除する機能、メッセージバッファに対してメッセージを送信/受信する機能、メッセージバッファの状態を参照する機能が含まれます。メッセージバッファはID番号で識別されるオブジェクトであり、メッセージバッファのID番号をメッセージバッファIDと呼びます。

メッセージバッファは、メッセージの送信を待つタスクの待ち行列(送信待ち行列)とメッセージの受信を待つタスクの待ち行列(受信待ち行列)を持ちます。また、送信されたメッセージを格納するためのメッセージバッファ領域を持ちます。メッセージを送信する側(イベントを知らせる側)では、送信したいメッセージをメッセージバッファにコピーします。メッセージバッファ領域の空き領域が足りなくなった場合、メッセージバッファ領域に十分な空きができるまでメッセージバッファへの送信待ち状態になります。

メッセージバッファへの送信待ち状態になったタスクは、そのメッセージバッファの送信待ち行列につながれます。一方、メッセージを受信する側(イベントを待つ側)では、メッセージバッファに入っているメッセージを一つ取り出します。メッセージバッファにメッセージが入っていない場合は、次にメッセージが送られてくるまでメッセージバッファからの受信待ち状態になります。メッセージバッファからの受信待ち状態になったタスクは、そのメッセージバッファの受信待ち行列につながれます。

メッセージバッファ領域のサイズを0にすることで、同期メッセージ機能を実現することができます。すなわち、送信側のタスクと受信側のタスクが、それぞれ相手のタスクがシステムコールを呼び出すのを待ち合わせ、両者がシステムコールを呼出した時点で、メッセージの受渡しが行われます。

メッセージバッファの生成情報

pk_cmbfの内容

```
typedef struct t_cmbf {
    VP    exinf;     /* Extended information      拡張情報                 */
    ATR   mbfatr;    /* Message buffer attribute  メッセージバッファ属性   */
    W     bufsz;     /* Message buffer size       メッセージバッファのサイズ */
    INT   maxmsz;    /* Maximum length of message メッセージの最大長       */
    UB    dsname[8]; /* Object name               DSオブジェクト名称       */
    VP    bufptr;    /* User buffer               ユーザーバッファポインタ */
} T_CMBF;
```

mbfatr := (TA_TFIFO ∥ TA_TPRI) | [TA_USERBUF] | [TA_DSNAME]

TA_TFIFO	0x00000000	送信待ちタスクのキューイングは FIFO
TA_TPRI	0x00000001	送信待ちタスクのキューイングは優先度順
TA_USERBUF	0x00000020	メッセージバッファの領域としてユーザが指定した領域を使用する
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する

メッセージバッファの生成情報

TA_TFIFO、TA_TPRI では、バッファが一杯の場合にメッセージを送信するタスクがメッセージバッファの待ち行列に並ぶ際の並び方を指定することができます。属性がTA_TFIFOであればタスクの待ち行列はFIFOとなり、属性がTA_TPRIであればタスクの待ち行列はタスクの優先度順となります。なお、メッセージキューの順序はFIFOのみです。

メッセージ受信待ちのタスクの待ち行列の順序はFIFOのみです。

TA_USERBUFを指定した場合にbufptrが有効になり、bufptrを先頭とするbufszバイトのメモリ領域をメッセージバッファ領域として使用します。この場合、メッセージバッファ領域はOSで用意しません。TA_USERBUFを指定しなかった場合は、bufptrは無視され、メッセージバッファ領域はOSが確保します。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

メッセージバッファの生成例

```
 1: #include <tk/tkernel.h>
 2: ID mbfid;                                    // メッセージバッファID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CMTX t_cmbf;
 7:     t_cmbf.mbfatr = TA_TPRI;                 // 送信待ちタスクは優先度順
 8:     t_cmbf.bufsz  = 1024;                    // バッファサイズは1024バイト
 9:     t_cmbf.maxmsz = 16;                      // メッセージの最大長は16バイト
10:     mbfid = tk_cre_mbf( &t_cmbf );           // tk_cre_mbfにより生成
11:     return mbfid > E_OK;                     // エラーコードのチェック
12: }
```

メッセージバッファの生成例

2行目： メッセージバッファ ID格納用の変数を外部変数として宣言します。

6行目： メッセージバッファ生成に必要なパラメータパケット（t_cmbf）を宣言します。

7～9行目： パラメータパケットに生成するメッセージバッファの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： メッセージバッファの属性をmbfatrに設定します。TA_TPRIを設定し、送信待ちタスクのキューイングを優先度順とます。さらにTA_USERBUFを指定していないため、bufptrは無効な項目となり、メッセージバッファ領域はOSが確保します。

8行目： メッセージバッファのサイズをbufszに設定します。

9行目： メッセージの最大長をmaxszに設定します。

10行目： tk_cre_mbfシステムコールでメッセージバッファを生成し、リターンコードを変数mbfidに格納します。

11行目： メッセージバッファが正常に生成できたかどうかを比較演算子で調べ、その結果を返します。

メッセージバッファのコーディング例

タスクA

```
 1: #include <tk/tkernel.h>
 2: #include <string.h>
 3: IMPORT ID mbfid;                          // メッセージバッファID番号格納用変数
 4:
 5: EXPORT void tsk_a(INT stacd, VP exinf)
 6: {
 7: char *p = "RENESAS";                                 // 送信データ
 8:     tk_snd_mbf( mbfid, p, strlen( p ), TMO_FEVR );
 9:     tk_ext_tsk( );                                   // tk_snd_mbfでデータを送信
10: }
```

タスクB

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID mbfid;                          // メッセージバッファID番号格納用変数
 3:
 4: EXPORT void tsk_b(INT stacd, VP exinf)
 5: {
 6: char buf[16];                                        // 受信バッファ
 7: INT  len;                                            // 受信データ長
 8:     while( 1 )  {                                    // tk_rcv_mbfでデータを受信
 9:         len = tk_rcv_mbf( mbfid, buf, TMO_FEVR );
10:         // 受信したデータに対する処理
11:     }
12: }
```

タスクA

3行目：上記の例ではメッセージバッファのID番号は変数mbfidに格納されていると仮定しています。
7行目：送信データの文字列を指すポインタ変数の宣言です。
8行目：TMO_FEVRの永久待ちでtk_snd_mbfシステムコールを発行し、データを送信します。
9行目：データ送信後はtk_ext_tskを発行し、タスクを終了します。

タスクB

3行目：上記の例ではメッセージバッファのID番号は変数mbfidに格納されていると仮定しています。
6行目：受信データ用のバッファを宣言します。上記の例では使用するメッセージバッファのメッセージ最大長を16バイトと仮定しています。
7行目：受信データのサイズを格納する変数を宣言します。
9行目：TMO_FEVRの永久待ちでtk_rcv_mbfシステムコールを発行し、データを受信します。
10行目：受信したデータに対する処理を実行します。

3.4.3　ランデブポート

ランデブポートのシステムコール一覧

■ランデブポート生成
```
ID porid   = tk_cre_por( T_CPOR *pk_cpor );
```
■ランデブポート削除
```
ER ercd    = tk_del_por( ID porid );
```
■ランデブ呼出
```
INT rmsgsz = tk_cal_por( ID porid, UINT calptn, VP msg, INT cmsgsz, TMO tmout );
```
■ランデブ受付
```
INT cmsgsz = tk_acp_por( ID porid, UINT acpptn, RNO *p_rdvno, VP msg, TMO tmout );
```
■ランデブ回送
```
ER ercd    = tk_fwd_por( ID porid, UINT calptn, RNO rdvno, VP msg, INT cmsgsz );
```
■ランデブ返答
```
ER ercd    = tk_rpl_rdv( RNO rdvno, VP msg, INT rmsgsz );
```
■ランデブポート状態参照
```
ER ercd    = tk_ref_por( ID porid, T_RPOR *pk_rpor );
```

●μT-Kernel仕様におけるランデブポートの特徴
- ランデブ呼出待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- ランデブ受付待ちタスクキューの並び方はFIFO順のみ

●tk_cal_por(***, ***, ***, ***, TMO_FEVR)がμITRON4.0仕様の cal_porに対応

●tk_acp_por(***, ***, ***, ***, TMO_FEVR)がμITRON4.0仕様の acp_porに対応
tk_acp_por(***, ***, ***, ***, TMO_POL) がμITRON4.0仕様のpacp_porに対応

ランデブ機能は、タスク間で同期通信を行うための機能で、あるタスクから別のタスクへの処理依頼と、後者のタスクから前者のタスクへの処理結果の返却を一連の手順としてサポートします。双方のタスクが待ち合わせるためのオブジェクトをランデブポートと呼びます。ランデブ機能は、典型的にはクライアント / サーバモデルのタスク間通信を実現するために用いられるが、クライアント / サーバモデルよりも柔軟な同期通信モデルを提供するものです。

ランデブ機能には、ランデブポートを生成/削除する機能、ランデブポートに対して処理の依頼を行う機能(ランデブの呼出し)、ランデブポートで処理依頼を受け付ける機能(ランデブの受付)、処理結果を返す機能(ランデブの終了)、受け付けた処理依頼を他のランデブポートに回送する機能(ランデブの回送)ランデブポートおよびランデブの状態を参照する機能が含まれます。ランデブポートはID番号で識別されるオブジェクトであり、ランデブポートのID番号をランデブポートIDと呼びます。

ランデブポートに対して処理依頼を行う側のタスク(クライアント側のタスク)は、ランデブポートとランデブ条件、依頼する処理に関する情報を入れたメッセージ(これを呼出しメッセージと呼ぶ)を指定して、ランデブの呼出しを行います。一方、ランデブポートで処理依頼を受け付ける側のタスク(サーバ側のタスク)は、ランデブポートとランデブ条件を指定して、ランデブの受付を行います。

ランデブ条件は、ビットパターンで指定します。あるランデブポートに対して、呼出したタスクのランデブ条件のビットパターンと、受け付けたタスクのランデブ条件のビットパターンをビット毎に論理積をとり、結果が 0 以外の場合にランデブが成立します。ランデブを呼出したタスクは、ランデブが成立するまでランデブ呼出し待ち状態となります。逆に、ランデブを受け付けるタスクは、ランデブが成立するまでランデブ受付待ち状態となります。

ランデブが成立すると、ランデブを呼出したタスクから受け付けたタスクへ、呼出しメッセージが渡されます。ランデブを呼出したタスクはランデブ終了待ち状態へ移行し、依頼した処理が完了するのを待ちます。一方、ランデブを受け付けたタスクは待ち解除され、依頼された処理を行います。ランデブを受け付けたタスクが依頼された処理を完了すると、処理結果を返答メッセージの形で呼出したタスクに渡し、ランデブを終了します。この時点で、ランデブを呼出したタスクが、ランデブ終了待ち状態から待ち解除さます。

ランデブポートは、ランデブ呼出し待ち状態のタスクをつなぐための呼出し待ち行列と、ランデブ受付待ち状態のタスクをつなぐための受付待ち行列を持ちます。それに対して、ランデブが成立した後は、ランデブした双方のタスクはランデブポートから切り離されます。すなわち、ランデブポートは、ランデブ終了待ち状態のタスクをつなぐための待ち行列は持ちません。また、ランデブを受け付け、依頼された処理を実行しているタスクに関する情報も持ちません。

カーネルは、同時に成立しているランデブを識別するために、オブジェクト番号を付与する。ランデブのオブジェクト番号をランデブ番号と呼びます。ランデブ番号の付与方法は実装定義であるが、少なくとも、ランデブを呼出したタスクを指定するための情報を含んでいなければなりません。また、同じタスクが呼出したランデブであっても、1回目のランデブと2回目のランデブで異なるランデブ番号を付与するなど、できる限りユニークにしなければなりません。

【仕様決定の理由】

ランデブ機能は、他の同期・通信機能を組み合わせて実現することも可能であるが、返答を伴う通信を行う場合には、それ専用の機能を用意した方が、アプリケーションプログラムが書きやすく、他の同期・通信機能を組み合わせるよりも効率を上げることができます。一例として、ランデブ機能は、メッセージの受渡しが終わるまで双方のタスクを待たせておくために、メッセージを格納するための領域が必要ないという利点があります。

同じタスクが呼出したランデブであっても、ランデブ番号をできる限りユニークにしなければならないのは、次の理由によります。ランデブが成立してランデブ終了待ち状態となっているタスクが、タイムアウトや待ち状態の強制解除などにより待ち解除された後、再度ランデブを呼出してランデブが成立した場合を考えます。この時、最初のランデブのランデブ番号と、後のランデブのランデブ番号が同一の値であると、最初のランデブを終了させようとした時に、ランデブ番号が同一であるために後のランデブが終了してしまいます。２つのランデブに異なるランデブ番号を付与し、ランデブ終了待ち状態のタスクに待ち対象のランデブ番号を記憶しておけば、最初のランデブを終了させようとした時にエラーとすることができます。

ランデブポートの生成情報

pk_porの内容

```
typedef struct t_cpor {
   VP    exinf;      /* Extended information             拡張情報                  */
   ATR   poratr;     /* Port attribute                   ランデブポート属性        */
   INT   maxcmsz;    /* Maximum length of call message   呼出時のメッセージの最大長 */
   INT   maxrmsz;    /* Maximum length of replay message 返答時のメッセージの最大長 */
   UB    dsname[8];  /* Object name                      DS オブジェクト名称       */
} T_CPOR;
```

poratr := (TA_TFIFO // TA_TPRI) | [TA_DSNAME]

TA_TFIFO	0x00000000	呼出し待ちタスクのキューイングはFIFO
TA_TPRI	0x00000001	呼出し待ちタスクのキューイングは優先度順
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する

ランデブポートの生成情報

TA_TFIFO、TA_TPRI では、ランデブ呼出し待ちのタスクの待ち行列の並び順を指定します。ランデブ受け付け待ちのタスクの待ち行列はFIFOのみです。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

ランデブポートの生成例

```
 1: #include <tk/tkernel.h>
 2: ID porid;                                    // ランデブポートID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CPOR t_cpor;
 7:     t_cpor.poratr  = TA_TPRI;                // 待ちタスクキューは優先度順
 8:     t_cpor.maxcmsz = 1024;                   // 呼出時のメッセージの最大長
 9:     t_cpor.maxrmsz = 1024;                   // 返答時のメッセージの最大長
10:     porid = tk_cre_por( &t_cpor );           // tk_cre_porにより生成
11:     return porid > E_OK;                     // エラーコードのチェック
12: }
```

ランデブポートの生成例

2行目： ランデブポートID格納用の変数を外部変数として宣言します。

6行目： ランデブポート生成に必要なパラメータパケット（t_cpor）を宣言します。

7～9行目： パラメータパケットに生成するランデブポートの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： ランデブポートの属性をporatrに設定します。TA_TPRIを設定し、呼出し待ちタスクのキューイングを優先度順とます。

8行目： 呼出時のメッセージの最大長をmaxcmszに設定します。

9行目： 返答時のメッセージの最大長をmaxrmszに設定します。

10行目： tk_cre_porシステムコールでランデブポートを生成し、リターンコードを変数poridに格納します。

11行目： ランデブポートが正常に生成できたかどうかを比較演算子で調べ、その結果を返します。

3.5　メモリプール
3.5.1　固定長メモリプール
固定長メモリプールのシステムコール一覧

■固定長メモリプール生成
```
ID mpfid = tk_cre_mpf( T_CMPF *pk_cmpf );
```
■固定長メモリプール削除
```
ER ercd  = tk_del_mpf( ID mpfid );
```
■固定長メモリブロック獲得
```
ER ercd  = tk_get_mpf( ID mpfid, VP *p_blf, TMO tmout );
```
■固定長メモリブロック返却
```
ER ercd  = tk_rel_mpf( ID mpfid, VP blf );
```
■固定長メモリプール状態参照
```
ER ercd  = tk_ref_mpf( ID mpfid, T_RMPF *pk_rmpf );
```

●μT-Kernel仕様における固定長メモリプールの特徴
- 待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- 保護レベルは全て保護レベル0相当として扱う（T-Kernel仕様との整合性が目的）

●tk_get_mpf(***, ***, TMO_FEVR)がμITRON4.0仕様の get_mpfに対応
tk_get_mpf(***, ***, TMO_POL) がμITRON4.0仕様のpget_mpfに対応

固定長メモリプールは、固定されたサイズのメモリブロックを動的に管理するためのオブジェクトです。固定長メモリプール機能には、固定長メモリプールを生成/削除する機能、固定長メモリプールに対してメモリブロックを獲得/返却する機能、固定長メモリプールの状態を参照する機能が含まれます。固定長メモリプールはID番号で識別されるオブジェクトであり、固定長メモリプールのID番号を固定長メモリプールIDと呼びます。

固定長メモリプールは、固定長メモリプールとして利用するメモリ領域（これを固定長メモリプール領域、または単にメモリプール領域と呼ぶ）と、メモリブロックの獲得を待つタスクの待ち行列を持ちます。固定長メモリプールからメモリブロックを獲得するタスクは、メモリプール領域に空きがなくなった場合、次にメモリブロックが返却されるまで固定長メモリブロックの獲得待ち状態となります。固定長メモリブロックの獲得待ち状態になったタスクは、その固定長メモリプールの待ち行列につながれます。

【T-Kernel仕様との差異】

μT-Kernel仕様ではMMUの無いシステムを前提としているため、保護レベルに関する機能はサポートしていません。それにも関わらず保護レベル(TA_RNGn)の属性が残っているのは、T-Kernel仕様との互換性を保つためです。

固定長メモリプールの生成情報

pk_cmpfの内容

```
typedef struct t_cmpf {
    VP    exinf;       /* Extended information         拡張情報                  */
    ATR   mpfatr;      /* Memory pool attribute        固定長メモリプール属性    */
    W     mpfcnt;      /* Number of blocks in whole memory pool                  */
                       /*                    固定長メモリプール全体のブロック数 */
    W     blfsz;       /* Fixed size memory block size メモリブロックサイズ      */
    UB    dsname[8];   /* Object name                  DSオブジェクト名称        */
    VP    bufptr;      /* User buffer                  ユーザーバッファポインタ  */
} T_CMPF;
```

```
mpfatr := (TA_TFIFO // TA_TPRI) | [TA_USERBUF] | [TA_DSNAME] |
          (TA_RNG0 // TA_RNG1 // TA_RNG2 // TA_RNG3)
```

TA_TFIFO	0x00000000	メモリ獲得待ちタスクのキューイングはFIFO
TA_TPRI	0x00000001	メモリ獲得待ちタスクのキューイングは優先度順
TA_USERBUF	0x00000020	メモリプール領域としてユーザが指定した領域を使用する
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する
TA_RNG0	0x00000000	メモリのアクセス制限を保護レベル0とする
TA_RNG1	0x00000100	メモリのアクセス制限を保護レベル1とする
TA_RNG2	0x00000200	メモリのアクセス制限を保護レベル2とする
TA_RNG3	0x00000300	メモリのアクセス制限を保護レベル3とする

固定長メモリプールの生成情報

TA_TFIFO、TA_TPRI では、タスクがメモリ獲得のためにメモリプールの待ち行列に並ぶ際の並び方を指定することができます。属性がTA_TFIFOであればタスクの待ち行列はFIFOとなり、属性がTA_TPRIであればタスクの待ち行列はタスクの優先度順となります。

TA_RNGnでは、メモリのアクセスを制限する保護レベルを指定します。しかしμT-KernelではMMUの無いシステムを前提としているため、全て保護レベル0相当として動作します。

TA_USERBUFを指定した場合にbufptrが有効になり、bufptrを先頭とするmpfcnt*blfszバイトのメモリ領域をメモリプール領域として使用します。この場合、メモリプール領域はOSで用意しません。TA_USERBUFを指定しなかった場合は、bufptrは無視され、メモリプール領域はOSが確保します。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコール からのみ操作可能です。

固定長メモリプールの生成例

```
 1: #include <tk/tkernel.h>
 2: ID mpfid;                                  // 固定長メモリプールID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CMPF t_cmpf;
 7:     t_cmpf.mpfatr = TA_TPRI;               // 待ちタスクキューは優先度順
 8:     t_cmpf.mpfcnt = 6;                     // ブロック数は６個
 9:     t_cmpf.blfsz  = 12;                    // メモリブロックサイズは12バイト
10:     mpfid = tk_cre_mpf( &t_cmpf );        // tk_cre_mpfにより生成
11:     return mpfid > E_OK;                  // エラーコードのチェック
12: }
```

固定長メモリプールの生成例

2行目： 固定長メモリプールID格納用の変数を外部変数として宣言します。

6行目： 固定長メモリプール生成に必要なパラメータパケット（t_cmpf）を宣言します。

7～9行目： パラメータパケットに生成する固定長メモリプールの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： 固定長メモリプールの属性をmpfatrに設定します。TA_TPRIを設定し、メモリ獲得待ちタスクのキューイングを優先度順とます。また、保護レベルは何もしていないため、TA_RNG0の保護レベル０を指定したことになります。さらにTA_USERBUFを指定していないため、bufptrは無効な項目となり、固定長メモリプール領域はOSが確保します。

8行目： 固定長メモリプール全体のブロック数をmpfcntに設定します。

9行目： メモリブロックサイズをblfszに設定します。

10行目： tk_cre_mpfシステムコールで固定長メモリプールを生成し、リターンコードを変数mpfidに格納します。

11行目： 固定長メモリプールが正常に生成できたかどうかを比較演算子で調べ、その結果を返します。

固定長メモリプールのコーディング例（１）

ユーザ固有のメッセージパケット（user.h）

```
 1: typedef struct user_msg {
 2:     T_MSG    msghead;              // メッセージキュー（OS管理エリア）
 3:     ID       mpfid;                // 固定長メモリプールID
 4:     char    *data;                 // 送信データの先頭アドレス
 5: } USER_MSG;
```

タスクＡ

```
 1: #include <tk/tkernel.h>
 2: #include "user.h"                          // ユーザ固有ヘッダ
 3: IMPORT ID mbxid, mpfid;                    // 各種ID番号格納用変数
 4:
 5: EXPORT void tsk_a(INT stacd, VP exinf)
 6: {
 7: USER_MSG *msg;                            // メッセージパケットのポインタ
 8:     tk_get_mpf( mpfid, (VP *)&msg, TMO_FEVR );
 9:     msg->mpfid = mpfid;          // tk_get_mpfで固定長メモリブロックを獲得
10:     msg->data = "RENESAS";                // 送信データを設定
11:     tk_snd_mbx( mbxid, (T_MSG *)msg );    // tk_snd_mbxでメッセージを送信
12:     tk_ext_tsk( );
13: }
```

メッセージパケット（user.h）

1～5行目： ユーザ固有のメッセージパケットの型宣言です。
2行目： メッセージのキューイングがTA_MFIFOの場合、T_MSG型のmsgheadメンバを設けます。
3行目： メッセージパケット用に使用する固定長メモリプールのIDを格納するmpfidメンバを設けます。
4行目： 送信データのアドレスを設定するdataメンバを設けます。
5行目： 上記のメッセージパケットをUSER_MSG型として宣言します。

タスクＡ

3行目： 上記の例ではメールボックスのID番号は変数mbxid、固定長メモリプールのID番号は変数mpfidに格納されていると仮定しています。
7行目： メッセージパケットを指すmsgポインタ変数を宣言します。
8行目： TMO_FEVRの永久待ちでtk_get_mpfシステムコールを発行し、メッセージパケット用のメモリブロックを固定長メモリプールから獲得します。
9～10行目： 獲得したメッセージパケットに対して必要な情報を設定します。mpfidメンバにはメッセージパケットを獲得した固定長メモリプールのIDを設定します。
11行目： tk_snd_mbxシステムコールでメッセージパケットを送信します。
12行目： tk_ext_tskシステムコールでタスクを終了します。

固定長メモリプールのコーディング例（２）

タスクＢ

```
 1: #include <tk/tkernel.h>
 2: #include "user.h"                          // ユーザ固有ヘッダ
 3: IMPORT ID mbxid;                           // メールボックスID番号格納用変数
 4:
 5: EXPORT void tsk_b(INT stacd, VP exinf)
 6: {
 7: USER_MSG *msg;                             // メッセージパケットのポインタ
 8:     while( 1 )  {                          // tk_rcv_mbxでメッセージを受信
 9:         tk_rcv_mbx( mbxid, (T_MSG **)&msg, TMO_FEVR );
10:         //  受信したメッセージに対する処理
11:         tk_rel_mpf( msg->mpfid, msg ); // tk_rel_mpfでメモリブロックの返却
12:     }
13: }
```

タスクＢ

3行目：上記の例ではメールボックスのID番号は変数mbxidに格納されていると仮定しています。
7行目：メッセージパケットを指すmsgポインタ変数を宣言します。
9行目：tk_rcv_mbxシステムコールを発行し、メッセージの受信を待ちます。受信したメッセージの先頭アドレスはmsg変数に格納します。
10行目：受信したメッセージに対する処理を実行します。
11行目：受信したメッセージに対する処理の終了後はパラメータパケットをメモリブロックとして固定長メモリプールにtk_rel_mpfシステムコールで返却します。その際、固定長メモリプールのID番号は、パラメータパケットに格納されているmpfidメンバを使用します。

3.5.2　可変長メモリプール

可変長メモリプールのシステムコール一覧

■可変長メモリプール生成
```
ID mplid = tk_cre_mpl( T_CMPL *pk_cmpl );
```
■可変長メモリプール削除
```
ER ercd  = tk_del_mpl( ID mplid );
```
■可変長メモリブロック獲得
```
ER ercd  = tk_get_mpl( ID mplid, INT blksz, VP *p_blf, TMO tmout );
```
■可変長メモリブロック返却
```
ER ercd  = tk_rel_mpl( ID mplid, VP blf );
```
■可変長メモリプール状態参照
```
ER ercd  = tk_ref_mpl( ID mplid, T_RMPL *pk_rmpl );
```

●μT-Kernel仕様における可変長メモリプールの特徴
- 待ちタスクキューの並び方としてFIFO順(TA_TFIFO)、優先度順(TA_TPRI)が選択可能
- 保護レベルは全て保護レベル0相当として扱う（T-Kernel仕様との整合性が目的）

●tk_get_mpl(***, ***, ***, TMO_FEVR)がμITRON4.0仕様の get_mplに対応
tk_get_mpl(***, ***, ***, TMO_POL) がμITRON4.0仕様のpget_mplに対応

可変長メモリプールは、任意のサイズのメモリブロックを動的に管理するためのオブジェクトです。可変長メモリプール機能には、可変長メモリプールを生成／削除する機能、可変長メモリプールに対してメモリブロックを獲得／返却する機能、可変長メモリプールの状態を参照する機能が含まれます。可変長メモリプールはID番号で識別されるオブジェクトであり、可変長メモリプールのID番号を可変長メモリプールIDと呼びます。

可変長メモリプールは、可変長メモリプールとして利用するメモリ領域（これを可変長メモリプール領域、または単にメモリプール領域と呼ぶ）と、メモリブロックの獲得を待つタスクの待ち行列を持ちます。可変長メモリプールからメモリブロックを獲得するタスクは、メモリプール領域の空き領域が足りなくなった場合、十分なサイズのメモリブロックが返却されるまで可変長メモリブロックの獲得待ち状態となります。可変長メモリブロックの獲得待ち状態になったタスクは、その可変長メモリプールの待ち行列につながれます。

可変長メモリプールの生成情報

pk_cmplの内容

```
typedef struct t_cmpl {
    VP    exinf;       /* Extended information  拡張情報                          */
    ATR   mplatr;      /* Memory pool attribute 可変長メモリプール属性            */
    W     mplsz;       /* Size of whole memory pool (byte)                       */
                       /*                        メモリプール全体のサイズ(バイト数) */
    UB    dsname[8];   /* Object name           DSオブジェクト名称                */
    VP    bufptr;      /* User buffer           ユーザーバッファポインタ          */
} T_CMPL;
```

```
mplatr := (TA_TFIFO // TA_TPRI) | [TA_USERBUF] | [TA_DSNAME] |
          (TA_RNG0 // TA_RNG1 // TA_RNG2 // TA_RNG3)
```

TA_TFIFO	0x00000000	メモリ獲得待ちタスクのキューイングはFIFO
TA_TPRI	0x00000001	メモリ獲得待ちタスクのキューイングは優先度順
TA_USERBUF	0x00000020	メモリプール領域としてユーザが指定した領域を使用する
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する
TA_RNG0	0x00000000	メモリのアクセス制限を保護レベル0とする
TA_RNG1	0x00000100	メモリのアクセス制限を保護レベル1とする
TA_RNG2	0x00000200	メモリのアクセス制限を保護レベル2とする
TA_RNG3	0x00000300	メモリのアクセス制限を保護レベル3とする

可変長メモリプールの生成情報

TA_TFIFO、TA_TPRI では、タスクがメモリ獲得のためにメモリプールの待ち行列に並ぶ際の並び方を指定することができます。属性がTA_TFIFOであればタスクの待ち行列はFIFOとなり、属性がTA_TPRIであればタスクの待ち行列はタスクの優先度順となります。

タスクがメモリ獲得待ちの行列を作った場合は、待ち行列先頭のタスクに優先してメモリを割当てます。待ち行列の２番目以降により少ないメモリサイズを要求しているタスクがあった場合も、そのタスクが先にメモリを獲得することはありません。例えば、ある可変長メモリプールに対して要求メモリサイズ＝400 のタスクAと要求メモリサイズ＝100のタスクBがこの順で待っており、別のタスクのtk_rel_mplによりメモリサイズ＝200の連続空きメモリ領域ができたとします。このとき、行列の先頭ではないが要求サイズの少ないタスク B が先にメモリを獲得することはありません。

TA_RNGnでは、メモリのアクセスを制限する保護レベルを指定します。しかしμT-KernelではMMUの無いシステムを前提としているため、全て保護レベル0相当として動作します。

TA_USERBUF を指定した場合に bufptr が有効になり、bufptr を先頭とする mplsz バイトのメモリ領域をメモリプール領域として使用します。この場合、メモリプール領域はOSで用意しません。TA_USERBUFを指定しなかった場合は、bufptrは無視され、メモリプール領域はOSが確保します。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

可変長メモリプール生成例

```
 1: #include <tk/tkernel.h>
 2: ID mplid;                                  // 可変長メモリプールID格納用変数
 3:
 4: EXPORT BOOL create(void)
 5: {
 6: T_CMPL t_cmpl;
 7:     t_cmpl.mplatr = TA_TPRI;               // 待ちタスクキューは優先度順
 8:     t_cmpl.mplsz  = 4096;                  // メモリプール全体は４kバイト
 9:     mplid = tk_cre_mpl( &t_cmpl );         // tk_cre_mplにより生成
10:     return mplid > E_OK;                   // エラーコードのチェック
11: }
```

可変長メモリプールの生成例

2行目： 可変長メモリプールID格納用の変数を外部変数として宣言します。

6行目： 可変長メモリプール生成に必要なパラメータパケット（t_cmpl）を宣言します。

7～8行目： パラメータパケットに生成する可変長メモリプールの情報を設定します。exinfの拡張情報とdsnameのオブジェクト名称は省略可能であるため、指定しません。

7行目： 可変長メモリプールの属性をmplatrに設定します。TA_TPRIを設定し、メモリ獲得待ちタスクのキューイングを優先度順とます。また、保護レベルは何もしていないため、TA_RNG0の保護レベル０を指定したことになります。さらにTA_USERBUFを指定していないため、bufptrは無効な項目となり、固定長メモリプール領域はOSが確保します。

8行目： 可変長メモリプール全体のサイズをmplszに設定します。

9行目： tk_cre_mplシステムコールで可変長メモリプールを生成し、リターンコードを変数mplidに格納します。

10行目： 可変長メモリプールが正常に生成できたかどうかを比較演算子で調べ、その結果を返します。

可変長メモリプールのコーディング例

タスク

```
 1: #include <tk/tkernel.h>
 2: IMPORT ID mblid;                          // 可変長メモリプールID番号格納用変数
 3:
 4: EXPORT void tsk(INT stacd, VP exinf)
 5: {
 6: char *buffer;                             // バッファアドレス用ポインタ
 7:     tk_get_mpl( mplid, 1024, (VP *)&buffer, TMO_FEVR );
 8:                                           // tk_get_mplでメモリブロックを獲得
 9:     //  獲得したメモリブロックを使用した処理
10:     tk_rel_mpl( mplid, buffer );          // tk_rel_mplでメモリブロックの返却
11:     tk_ext_tsk( );                        // tk_ext_tskで終了
12: }
```

タスク

2行目：上記の例では可変長メモリプールのID番号は変数mplidに格納されていると仮定しています。
6行目：メモリブロックを指すためのポインタ変数bufferを宣言します。
7行目：tk_get_mplシステムコールを発行し、メモリブロックを獲得します。獲得したメモリブロックの先頭アドレスはbuffer変数に格納します。
9行目：獲得したメモリブロックを使用して処理を実行します。
10行目：獲得したメモリブロックをtk_rel_mplシステムコールで返却します。

3.6　時間管理機能

3.6.1　システム時刻管理

システム時刻管理のシステムコール一覧

■システム時刻設定
```
ER ercd = tk_set_tim( SYSTIM *pk_tim );
```
■システム時刻参照
```
ER ercd = tk_get_tim( SYSTIM *pk_tim );
```
■システム稼働時間参照
```
ER ercd = tk_get_otm( SYSTIM *pk_tim );
```

●システム稼動時刻
- システム時刻とは異なり、システム起動時からの単純増加する稼働時間を表わす。
- tk_set_timによる時刻設定に影響されない。

●システム時刻と相対時刻
- 時刻を絶対値で指定する場合はシステム時刻(SYSTIM)を用いる。
- システムコールを呼出した時刻等からの相対値を指定する場合は相対時刻(RELTIM)を用いる。

●SYSTIM　システム時刻
基準時間１ms秒、64ビット符号付き整数
```
typedef struct systim {
        W  hi;   // 上位32ビット
        UW lo;   // 下位32ビット
} SYSTIM;
```

●RELTIM　相対時刻
基準時間１ms秒、32ビット符号なし整数(UW)
```
typedef UW RELTIM;
```
●TMO　タイムアウト時間
基準時間１ms秒、32ビット符号付き整数(W)
```
typedef W  TMO;
```

システム時刻管理機能は、システム時刻を操作するための機能です。システム時刻を設定／参照する機能、システム稼働時間を参照する機能が含まれます。

3.6.2　周期ハンドラ

周期ハンドラのシステムコール一覧

■周期ハンドラ生成
```
ID cycid = tk_cre_cyc( T_CCYC *pk_ccyc );
```
■周期ハンドラ削除
```
ER ercd  = tk_del_cyc( ID cycid );
```
■周期ハンドラの動作開始
```
ER ercd  = tk_sta_cyc( ID cycid );
```
■周期ハンドラの動作停止
```
ER ercd  = tk_stp_cyc( ID cycid );
```
■周期ハンドラ状態参照
```
ER ercd  = tk_ref_cyc( ID cycid, T_RCYC *pk_rcyc );
```

●μT-Kernel仕様における周期ハンドラの特徴
- 記述言語として高級言語(TA_HLNG)とアセンブリ言語(TA_ASM)が選択可能
- 生成時、起動するかどうかをTA_STAで指定可能
- 起動位相を保存するかどうかをTA_PHSで指定可能

●周期ハンドラの記述例
```
void cychdr( VP exinf )
{
                // 周期ハンドラに与えられた処理
    return ;
}
```

周期ハンドラは、一定周期で起動されるタイムイベントハンドラです。周期ハンドラ機能には、周期ハンドラを生成/削除する機能、周期ハンドラの動作を開始/停止する機能、周期ハンドラの状態を参照する機能が含まれます。周期ハンドラはID番号で識別されるオブジェクトであり、周期ハンドラのID番号を周期ハンドラIDと呼びます。

周期ハンドラの起動周期と起動位相は、周期ハンドラの生成時に、周期ハンドラ毎に設定することができます。カーネルは、周期ハンドラの操作時に設定された起動周期と起動位相から、周期ハンドラを次に起動すべき時刻を決定します。周期ハンドラの生成時には、周期ハンドラを生成した時刻に起動位相を加えた時刻を次に起動すべき時刻とします。周期ハンドラを起動すべき時刻になると、その周期ハンドラの拡張情報（exinf）をパラメータとして、周期ハンドラを起動します。またこの時、周期ハンドラの起動すべき時刻に起動周期を加えた時刻を、次に起動すべき時刻とします。また、周期ハンドラの動作を開始する時に、次に起動すべき時刻を決定しなおす場合があります。

周期ハンドラは、動作している状態か動作していない状態かのいずれかの状態をとります。周期ハンドラが動作していない状態の時には、周期ハンドラを起動すべき時刻となっても周期ハンドラを起動せず、次に起動すべき時刻の決定のみを行います。周期ハンドラの動作を開始するシステムコール（tk_sta_cyc）が呼び出されると、周期ハンドラを動作している状態に移行し、必要なら周期ハンドラを次に起動すべき時刻を決定します。周期ハンドラの動作を停止するシステムコール(tk_stp_cyc)が呼び出されると、周期ハンドラを動作していない状態に移行さます。周期ハンドラを生成した後にどちらの状態になるかは、周期ハンドラ属性によって決めることができます。

周期ハンドラの起動位相は、周期ハンドラを生成するシステムコールが呼び出された時刻を基準に、周期ハンドラを最初に起動する時刻を指定する相対時間と解釈します。周期ハンドラの起動周期は、周期ハンドラを（起動した時刻ではなく）起動すべきであった時刻を基準に、周期ハンドラを次に起動する時刻を指定する相対時間と解釈します。そのため、周期ハンドラが起動される時刻の間隔は、個々には起動周期よりも短くなる場合がありますが、長い期間で平均すると起動周期に一致します。

周期ハンドラの生成情報

pk_ccycの内容

```
typedef struct t_ccyc {
    VP      exinf;      /* Extended information    拡張情報              */
    ATR     cycatr;     /* Cycle handler attribute 周期ハンドラ属性      */
    FP      cychdr;     /* Cycle handler address   周期ハンドラアドレス  */
    RELTIM  cyctim;     /* Cycle interval          周期起動時間間隔      */
    RELTIM  cycphs;     /* Cycle phase             周期起動位相          */
    UB      dsname[8];  /* Object name             DSオブジェクト名称    */
} T_CCYC;
```

cycatr := (TA_ASM∥TA_HLNG) | [TA_STA] | [TA_PHS] | [TA_DSNAME]

TA_ASM	0x00000000	対象ハンドラがアセンブラで書かれている
TA_HLNG	0x00000001	対象ハンドラが高級言語で書かれている
TA_STA	0x00000002	周期ハンドラ生成後直ちに起動する
TA_PHS	0x00000004	起動位相を保存する
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する

周期ハンドラの生成情報

cychdrは周期ハンドラの先頭アドレス、cyctimは周期起動の時間間隔、cycphsは起動位相を表わします。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由して周期ハンドラを起動します。高級言語対応ルーチンによって、レジスタの退避と復帰が行われますから、単純な関数からのリターンによって終了します。

TA_ASM属性の場合の周期ハンドラの形式は実装定義とします。ただし、起動パラメータとしてexinfを渡さなければなりません。

cycphs は tk_cre_cyc によって周期ハンドラを生成してから最初の周期ハンドラの起動までの時間を表わします。その後は、cyctim間隔で周期起動を繰り返します。cycphsに0を指定した場合は、周期ハンドラの生成直後に周期ハンドラが起動されることになります。cyctimに0を指定することはできません。

周期ハンドラの n回目の起動は、周期ハンドラを生成してからcycphs + cyctim*(n-1)以上の時間が経過した後に行います。TA_STAを指定した場合は、周期ハンドラの生成時から周期ハンドラは動作状態となり、前述の時間隔で周期ハンドラが起動されます。TA_STAを指定しなかった場合は、起動周期の計測は行われるが、周期ハンドラは起動されません。

TA_PHSが指定されている場合は、tk_sta_cycによって周期ハンドラが活性化されても、起動周期はリセットされず前述のように周期ハンドラの生成時から計測している周期を維持します。TA_PHSが指定されていない場合は、tk_sta_cycによって起動周期がリセットされ、tk_sta_cyc呼出時からcyctim間隔で周期ハンドラが起動されます。

tk_sta_cycによるリセットでは、cycphsは適用されません。この場合、tk_sta_cycからn回目の周期ハンドラの起動は、tk_sta_cyc呼出時からcyctim*n以上の時間が経過した後となります。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

周期ハンドラの生成例

● 生成する周期ハンドラの例

```
EXPORT void cychdr( VP exinf )
{
  // 周期ハンドラに与えられた処理
}
```

```
 1: #include <tk/tkernel.h>
 2: IMPORT void cychdr(VP);
 3: ID cycid;
 4:
 5: EXPORT void create(void)
 6: {
 7: T_CCYC t_ccyc;
 8:     t_ccyc.exinf  = 0;              // 拡張情報の指定は自由
 9:     t_ccyc.cycatr = TA_HLNG;        // 高級言語を指定
10:     t_ccyc.cychdr = cychdr;         // 周期ハンドラの起動アドレス
11:     t_ccyc.cyctim = 10;             // 周期起動時間間隔の指定
12:     t_ccyc.cycphs = 0;              // 周期起動位相の指定
13:     cycid = tk_cre_cyc( &t_ccyc );  // tk_cre_cycシステムコールにより生成
14:     if( cycid > E_OK )              // エラーコードのチェック
15:         tk_sta_cyc( cycid );        // tk_sta_cycシステムコールで起動
16: }
```

周期ハンドラの生成例

3行目：周期ハンドラID格納用の変数を外部変数として宣言します。
7行目：周期ハンドラ生成に必要なパラメータパケット（t_ccyc）を宣言します。
8行目：周期ハンドラに渡す拡張情報をexinfに設定します。設定した値は周期ハンドラの引数に渡されます。
9行目：周期ハンドラの属性をcycatrに設定します。生成する周期ハンドラはC言語で記述されているためTA_HLNGを指定します。また、上記の例では周期ハンドラ生成後直ちに起動するためのTA_STAと起動位相を保存するTA_PHSは指定していません。
10行目：周期ハンドラの起動アドレスをcychdrに設定します。
11行目：周期ハンドラの周期起動時間間隔（10ms）をcyctimに設定します。
12行目：周期ハンドラの周期起動位相（0ms）をcycphsに設定します。
13行目：tk_cre_cycシステムコールで周期ハンドラを生成し、リターンコードを変数cycidに格納します。
14行目：周期ハンドラが正常に生成できたかどうかをif文でチェックします。正の値であれば正常に生成できているため、E_OK（ゼロ）より大きいかを調べます。
15行目：正常に生成できたのであれば、tk_sta_cycシステムコールで周期ハンドラを起動します。

3.6.3　アラームハンドラ

アラームハンドラのシステムコール一覧

■アラームハンドラ生成
```
ID almid = tk_cre_alm( T_CALM *pk_calm );
```
■アラームハンドラ削除
```
ER ercd  = tk_del_alm( ID almid );
```
■アラームハンドラの動作開始
```
ER ercd  = tk_sta_alm( ID almid, RELTIM almtim );
```
■アラームハンドラの動作停止
```
ER ercd  = tk_stp_alm( ID almid );
```
■アラームハンドラ状態参照
```
ER ercd  = tk_ref_alm( ID almid, T_RALM *pk_ralm );
```

●μT-Kernel仕様におけるアラームハンドラの特徴
－ 記述言語として高級言語(TA_HLNG)とアセンブリ言語(TA_ASM)が選択可能

●アラームハンドラの記述例
```
void almhdr( VP exinf )
{
                // アラームハンドラに与えられた処理
    return ;
}
```

アラームハンドラは、指定した時刻に起動されるタイムイベントハンドラです。アラームハンドラ機能には、アラームハンドラを生成/削除する機能、アラームハンドラの動作を開始/停止する機能、アラームハンドラの状態を参照する機能が含まれます。アラームハンドラはID番号で識別されるオブジェクトであり、アラームハンドラのID番号をアラームハンドラIDと呼びます。

アラームハンドラを起動する時刻(これをアラームハンドラの起動時刻と呼ぶ)は、アラームハンドラ毎に設定することができます。アラームハンドラの起動時刻になると、そのアラームハンドラの拡張情報(exinf)をパラメータとして、アラームハンドラを起動します。

アラームハンドラの生成直後には、アラームハンドラの起動時刻は設定されておらず、アラームハンドラの動作は停止しています。アラームハンドラの動作を開始するシステムコール（tk_sta_alm）が呼び出されると、アラームハンドラの起動時刻を、システムコールが呼び出された時刻から指定された相対時間後に設定します。アラームハンドラの動作を停止するシステムコール(tk_stp_alm)が呼び出されると、アラームハンドラの起動時刻の設定を解除します。また、アラームハンドラを起動する時にも、アラームハンドラの起動時刻の設定を解除し、アラームハンドラの動作を停止します。

アラームハンドラの生成情報

pk_calmの内容

```
typedef struct t_calm {
    VP    exinf;       /* Extended information    拡張情報                */
    ATR   almatr;      /* Alarm handler attribute アラームハンドラ属性    */
    FP    almhdr;      /* Alarm handler address   アラームハンドラアドレス */
    UB    dsname[8];   /* Object name             DSオブジェクト名称      */
} T_CALM;
```

cycalm := (TA_ASM∥TA_HLNG) | [TA_DSNAME]

TA_ASM	0x00000000	対象ハンドラがアセンブラで書かれている
TA_HLNG	0x00000001	対象ハンドラが高級言語で書かれている
TA_DSNAME	0x00000040	DSオブジェクト名称を指定する

アラームハンドラの生成情報

almhdrは起動されるアラームハンドラの先頭アドレスを表わします。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由してアラームハンドラを起動します。高級言語対応ルーチンによって、レジスタの退避と復帰が行われますから、単純な関数からのリターンによって終了します。

TA_ASM属性の場合、アラームハンドラの形式は実装定義とします。ただし、起動パラメータとしてexinfを渡さなければなりません。

TA_DSNAMEを指定した場合にdsnameが有効となり、DSオブジェクト名称として設定されます。DSオブジェクト名称はデバッガがオブジェクトを識別するために使用され、デバッガサポート機能のシステムコールからのみ操作可能です。

アラームハンドラの生成例

● 生成するアラームハンドラの例

```
EXPORT void almhdr( VP exinf )
{
  // アラームハンドラに与えられた処理
}
```

```
 1: #include <tk/tkernel.h>
 2: IMPORT void almhdr(VP);
 3: ID almid;
 4:
 5: EXPORT void create(void)
 6: {
 7: T_CALM t_calm;
 8:     t_calm.exinf  = 0;              // 拡張情報の指定は自由
 9:     t_calm.almatr = TA_HLNG;        // 高級言語を指定
10:     t_calm.almhdr = almhdr;         // アラームハンドラの起動アドレス
11:     almid = tk_cre_alm( &t_calm );  // tk_cre_almシステムコールにより生成
12:     if( almid > E_OK )              // エラーコードのチェック
13:         tk_sta_alm( almid, 100 );   // tk_sta_almシステムコールで起動
14: }                                   // 100ms後にアラームハンドラを起動
```

アラームハンドラの生成例

3行目：アラームハンドラID格納用の変数を外部変数として宣言します。

7行目：アラームハンドラ生成に必要なパラメータパケット（t_calm）を宣言します。

8行目：アラームハンドラに渡す拡張情報をexinfに設定します。設定した値はアラームハンドラの引数に渡されます。

9行目：アラームハンドラの属性をalmatrに設定します。生成するアラームハンドラはC言語で記述されているためTA_HLNGを指定します。

10行目：アラームハンドラの起動アドレスをalmhdrに設定します。

11行目：tk_cre_almシステムコールでアラームハンドラを生成し、リターンコードを変数almidに格納します。

12行目：アラームハンドラが正常に生成できたかどうかをif文でチェックします。正の値であれば正常に生成できているため、E_OK（ゼロ）より大きいかを調べます。

13行目：正常に生成できたのであれば、tk_sta_almシステムコールで100ms後にアラームハンドラを起動します。

3.7　割込み管理機能

3.7.1　割込みハンドラ管理

割込みハンドラ管理機能のシステムコール一覧

■割込みハンドラ定義

```
ER ercd = tk_def_int( UINT dintno, T_DINT *pk_dint );
```

■割込みハンドラから復帰

```
void  tk_ret_int( void );
```

tk_ret_intは高級言語対応ルーチンを使用した場合は呼出されることがない

●μT-Kernelにおける割込みハンドラの特徴
- 記述言語として高級言語(TA_HLNG)とアセンブリ言語(TA_ASM)が選択可能

●高級言語(TA_HLNG)属性の割込みハンドラの特徴
- 高級言語対応ルーチンを経由して起動される。
- レジスタの退避・復帰を行う必要はない。
- 関数の終了が割り込みハンドラの終了となる。
- 常に遅延ディスパッチが行われる。

●アセンブリ言語(TA_ASM)属性の割込みハンドラの特徴
- 原則として、起動時にはOSが介入しない。
- 出入口で使用するレジスタの退避・復帰を行う必要がある。
- tk_ret_intまたは割込みリターン命令で終了する。
- tk_ret_intを使用しない場合、遅延ディスパッチは行われない。

●割込みハンドラの記述例（高級言語対応ルーチン経由）

```
void inthdr( UINT dintno )
{
                // 割込みハンドラに与えられた処理
    return ;
}
```

割込みハンドラは、タスク独立部として扱われます。タスク独立部でも、タスク部と同じ形式でシステムコールを発行することが可能ですが、タスク独立部で発行できるシステムコールには以下のような制限があります。

・ 暗黙に自タスクを指定するシステムコールや自タスクを待ち状態にするシステムコールは発行することができません。その場合はE_CTX のエラーとなります。

タスク独立部の実行中はタスクの切り換え（ディスパッチ）は起こらず、システムコールの処理の結果ディスパッチの要求が出されても、タスク独立部を抜けるまでディスパッチが遅らされます。これを遅延ディスパッチ(delayed　dispatching)の原則と呼びます。

割り込みハンドラの定義情報

pk_dintの内容

```
typedef struct t_dint {
    ATR    intatr;  /* Interrupt handler attribute 割り込みハンドラ属性     */
    FP     inthdr;  /* Interrupt handler address   割り込みハンドラアドレス */
} T_DINT;
```

```
intatr := (TA_ASM∥TA_HLNG)

TA_ASM       0x00000000    対象ハンドラがアセンブラで書かれている
TA_HLNG      0x00000001    対象ハンドラが高級言語で書かれている
```

割り込みハンドラの定義情報

TA_ASM属性の場合、原則として、割込みハンドラの起動時にはOSが介入しません。割込み発生時には、CPUハードウェアの割込み処理機能により、このシステムコールで定義した割込みハンドラが直接起動されます。したがって、割込みハンドラの先頭と最後では、割込みハンドラで使用するレジスタの退避と復帰を行う必要があります。割込みハンドラは、tk_ret_intシステムコールまたはCPUの割込みリターン命令（またはそれに相当する手段）によって終了します。

tk_ret_intシステムコールを使用せずにOSを介すことなく割込みハンドラから復帰する手段は必須です。ただし、tk_ret_intシステムコールを使用しない場合は、遅延ディスパッチが行われなくても良いことになっています。tk_ret_intシステムコールによる割込みハンドラからの復帰も必須であり、この場合は遅延ディスパッチが行われなければなりません。

TA_HLNG 属性の場合は、高級言語対応ルーチンを経由して割込みハンドラを起動します。高級言語対応ルーチンによって、レジスタの退避と復帰が行われています。

RX62Nの割り込みベクタテーブル（可変）

優先順位	割り込み要求発生元	名称	ﾍﾞｸﾀ番号	ﾍﾞｸﾀｱﾄﾞﾚｽ ｵﾌｾｯﾄ	割り込みの検出方法	選択可能な割り込み要求先			IER	IPR
						CPU	DTC	DMAC		
高	BSC	BUSERR	16	0040h	レベル	○	×	×	IER02.IEN0	IPR00
↑	:	:	:	:	:	:	:	:	:	:
	FCU	FIFERR	21	0054h	レベル	○	×	×	IER02.IEN5	IPR01
		予約	22	0058h	－	×	×	×	IER02.IEN6	－
		FRDYI	23	005Ch	エッジ	○	×	×	IER02.IEN7	IPR02
	:	:	:	:	:	:	:	:	:	:
	CMT0	CMI0	28	0070h	エッジ	○	○	○	IER03.IEN4	IPR04
	CMT1	CMI1	29	0074h	エッジ	○	○	○	IER03.IEN5	IPR05
	CMT2	CMI2	30	0078h	エッジ	○	○	○	IER03.IEN6	IPR06
	CMT3	CMI3	31	007Ch	エッジ	○	○	○	IER03.IEN7	IPR07
	:	:	:	:	:	:	:	:	:	:
	ICU（外部端子）	IRQ0	64	0100h	エッジ/レベル	○	○	○	IER08.IEN0	IPR20
		IRQ1	65	0104h	エッジ/レベル	○	○	○	IER08.IEN1	IPR21
		IRQ2	66	0108h	エッジ/レベル	○	○	○	IER08.IEN2	IPR22
		IRQ3	67	010Ch	エッジ/レベル	○	○	○	IER08.IEN3	IPR23
		:	:	:	:	:	:	:	:	:
		IRQ13	77	0134h	エッジ/レベル	○	○	○	IER09.IEN5	IPR2D
		IRQ14	78	0138h	エッジ/レベル	○	○	○	IER09.IEN6	IPR2E
		IRQ15	79	013Ch	エッジ/レベル	○	○	○	IER09.IEN7	IPR2F
	:	:	:	:	:	:	:	:	:	:
	AD0	ADI0	98	0188h	エッジ	○	○	○	IER0C.IEN2	IPR44
	AD1	ADI1	99	018Ch	エッジ	○	○	○	IER0C.IEN3	IPR45
	:	:	:	:	:	:	:	:	:	:
	MTU0	TGIA0	114	01C8h	エッジ	○	○	○	IER0E.IEN2	IPR51
		TGIB0	115	01CCh	エッジ	○	○	×	IER0E.IEN3	
		TGIC0	116	01D0h	エッジ	○	○	×	IER0E.IEN4	
		TGID0	117	01D4h	エッジ	○	○	×	IER0E.IEN5	
		TCIV0	118	01D8h	エッジ	○	×	×	IER0E.IEN6	IPR52
		TGIE0	119	01DCh	エッジ	○	×	×	IER0E.IEN7	
↓		TGIF0	120	01E0h	エッジ	○	×	×	IER0F.IEN0	
低	:	:	:	:	:	:	:	:	:	:

割り込み定義番号（dintno）

RXファミリ用μT-Kernelの場合、割り込み定義番号はベクタ番号でインプリメントが行われています。
tk_def_intシステムコールの第１引数にはベクタ番号を渡すことになります。

割り込みハンドラの定義例

● 生成する割り込みハンドラの例

```
EXPORT void ad0_adi0_hdr( UNIT dintno )
{
  // 割り込みハンドラに与えられた処理
}
```

```
 1: #include <tk/tkernel.h>
 2: #include "iodefine.h"
 3: IMPORT void ad0_adi0_hdr(UNIT);
 4:
 5: EXPORT void create(void)
 6: {
 7: T_DINT t_dint;
 8:     t_dint.intatr = TA_HLNG;          // 高級言語を指定
 9:     t_dint.inthdr = ad0_adi0_hdr;    // 割り込みハンドラの起動アドレス
10:     tk_def_int( 98, &t_dint );        // tk_def_intシステムコールにより生成
11: //  tk_def_int( VECT( AD0, ADI0 ), &t_dint );        // VECTマクロの利用
12: }
```

割り込みハンドラの定義例

7行目：割り込みハンドラ定義に必要なパラメータパケット（t_dint）を宣言します。

8行目：割り込みハンドラの属性をintatrに設定します。生成する割り込みハンドラはC言語で記述されているためTA_HLNGを指定します。

9行目：割り込みハンドラの起動アドレスをinthdrに設定します。

10行目：tk_def_intシステムコールで割り込みハンドラを定義します。

11行目：tk_def_intシステムコールの第１引数で指定するベクタ番号はiodefine.hに定義されているVECTマクロを使用することが可能です。

3.7.2　CPU割込み制御

CPU割込み制御のライブラリ関数またはマクロ一覧

■DI(UINT intsts)
すべての外部割込みを禁止する。割込みを禁止する前の状態をintstsに保存する。

■EI(UINT intsts)
すべての外部割込みを許可する。正確には、intstsの状態に戻す。

■BOOL isDI(UINT intsts)
intsts に保存されている外部割込み禁止状態を調べる。
μT-Kernelで割込み禁止と判断される状態を割込み禁止状態とする。
戻値TRUE(0以外の値)：割込み禁止状態　FALSE：割込み許可状態

CPUの外部割込みフラグを制御します。一般的には、割込みコントローラに対しては何もしない。

割込み関係はハードウェア依存度が高く、システムごとに異なっているため共通化することが難しいため、割込み制御に関してはDI、EI、isDIのみを標準仕様として定めています。標準仕様とは別の機能を追加することも許されるが、その場合、関数名などは標準仕様と異なるものでなければなりません。

DI、EI、isDIは、ライブラリ関数またはC言語のマクロで提供され、これらはタスク独立部およびディスパッチ禁止・割込み禁止の状態から呼び出すことができます。

```
void foo(void)
{
UINT intsts;

    DI( intsts );
    if( isDI( intsts ) )  {
        /* この関数が呼び出された時点で既に割込み禁止であった */
    }
    else  {
        /* この関数が呼び出された時点では割込み許可であった */
    }
    EI( intsts );
}
```

3.8 システム状態管理機能

システム状態管理機能のシステムコール一覧

■タスクの優先順位の回転
```
ER ercd  = tk_rot_rdq( PRI tskpri );
```
■実行状態タスクのタスクID参照
```
ID tskid = tk_get_tid( void );
```
■ディスパッチ禁止
```
ER ercd  = tk_dis_dsp( void );
```
■ディスパッチ許可
```
ER ercd  = tk_ena_dsp( void );
```
■システム状態参照
```
ER ercd  = tk_ref_sys( T_RSYS *pk_rsys );
```
■バージョン参照
```
ER ercd  = tk_ref_ver( T_RVER *pk_rver );
```

システム状態管理機能は、システムの状態を変更 / 参照するための機能です。タスクの優先順位を回転する機能、実行状態のタスク ID を参照する機能、タスクディスパッチを禁止 / 解除する機能、コンテキストやシステム状態を参照する機能、カーネルのバージョンを参照する機能が含まれます。

【T-Kernel仕様との差異】

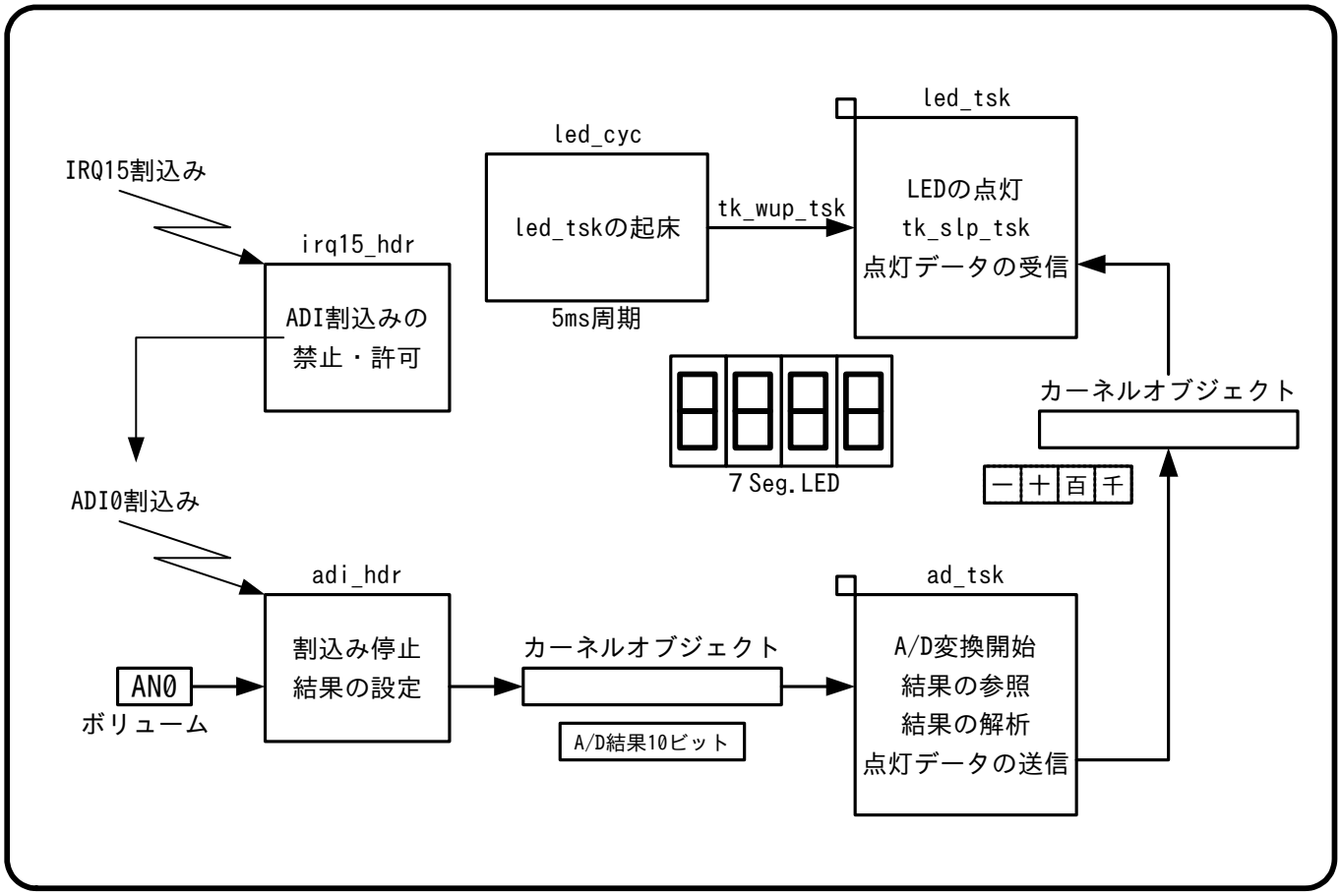
省電力モードを設定する機能（tk_set_pow）は存在しない。

第 4 章

マルチタスクシステム演習

4.1　システム構成

マルチタスクシステム演習のシステム構成



システム構成

マルチタスクシステム演習ではRX62N（RX600）を使用します。システム構成はAN0端子に接続されたボリュームの値を10進４桁で７セグメントLEDにダイナミック点灯します。また、A/D変換の開始・停止はIRQ15に接続したSW8のスイッチで制御可能とします。

タスク構成は既に決められており、ボリュームの値のA/D変換を行うad_tsk、７セグメントLEDへのダイナミック点灯を行うled_tskの２つのタスクでシステム全体を制御します。また、SW8のIRQ15割込みはirq15_hdr、A/D変換器のADI0割込みはadi_hdrの割込みハンドラが制御します。更にled_tskは７セグメントLEDのダイナミック点灯を扱うため、周期的な動作が必要です。そこで5msの間隔で起動されるled_cycの周期ハンドラからタスク付属同期機能のtk_wup_taskとtk_slp_taskの組み合わせで周期的な動作を実現します。

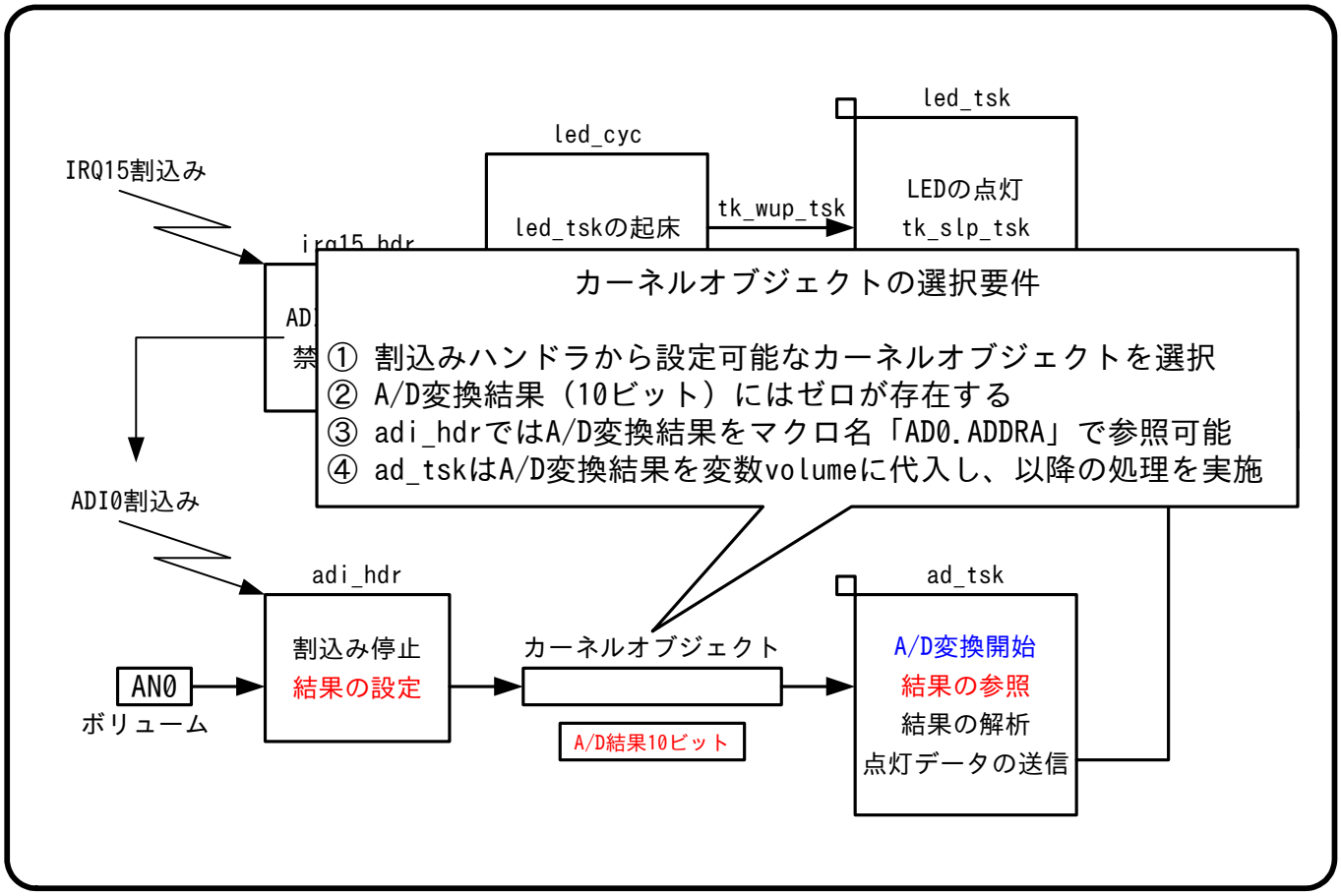
なお、IRQ15はは割込み優先レベル２、ADI0割込みは割込み優先レベル１を使用することにします。

マルチタスクシステム演習では

- adi_hdr割込みハンドラとad_tskタスクの間のカーネルオブジェクト
- ad_tskタスクとled_tskタスクの間のカーネルオブジェクト

の２つのカーネルオブジェクトを決定し、その部分のシステムコールのコーディングと使用するカーネルオブジェクトの生成を行います。次頁より、カーネルオブジェクトの決定における制約事項を説明します。

A/D変換結果の受け渡しに利用するカーネルオブジェクト



A/D変換結果の受け渡しに利用するカーネルオブジェクトの要件

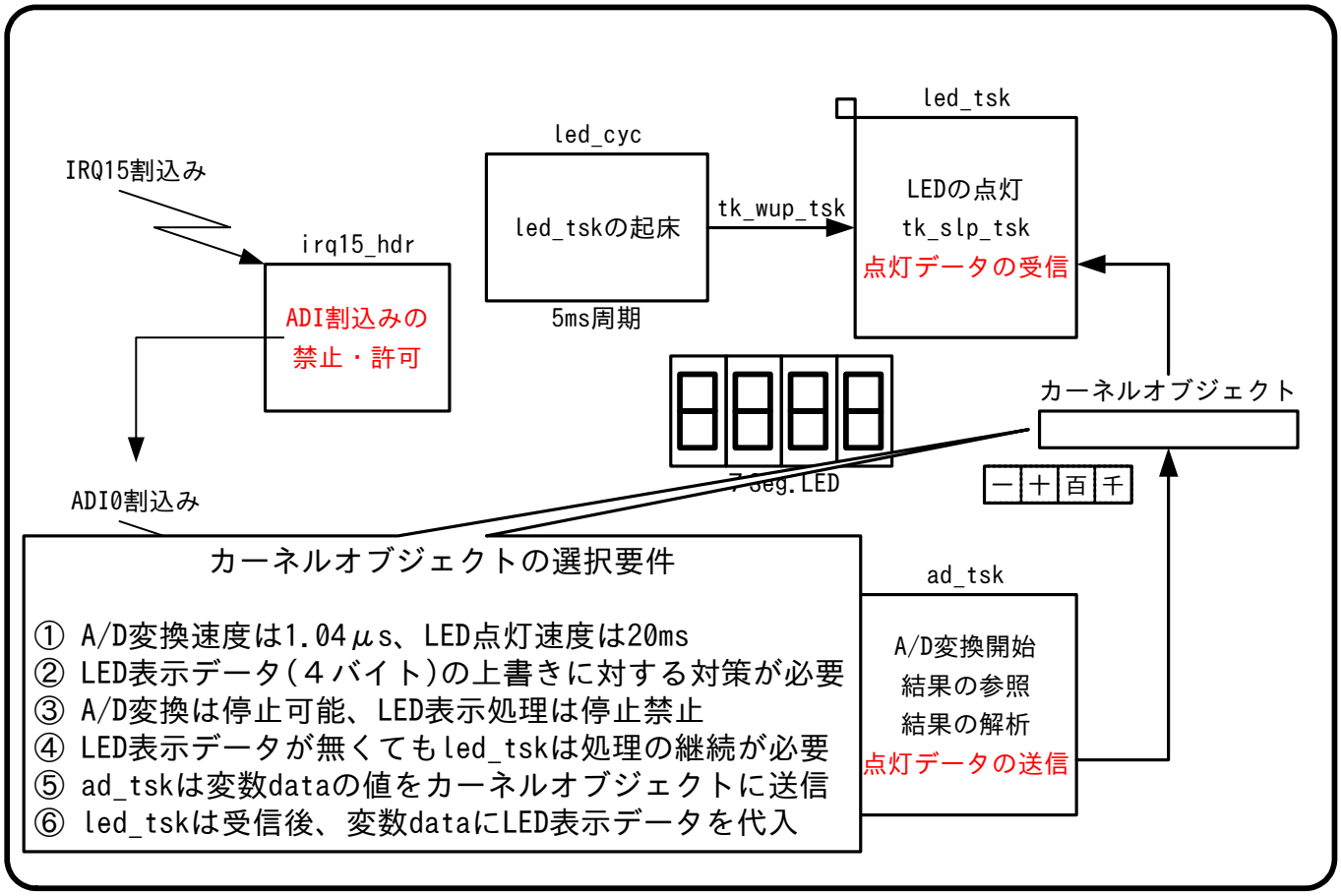
A/D変換の開始はad_taskタスクが行い、A/D変換の終了は割込み（ADI0）によって通知されます。従って、ad_taskタスクはA/D変換開始後、何らかのカーネルオブジェクトを使ってWAITING状態となります。逆にADI0割込みで起動されるadi_hdr割込みハンドラは、ad_taskタスクが発行したシステムコールと対のシステムコールを使い、変換結果と共にad_taskのWAITING状態を解除します。このため、ad_taskタスクとadi_hdr割込みハンドラの間で使用するカーネルオブジェクトには以下に示す２つの要件があります。

- 割込みハンドラからデータの設定が可能なカーネルオブジェクトであること
- 設定するA/D変換結果は10ビットであり、変換結果にはゼロが存在すること

上記２つの要件が満足されるのであれば、カーネルオブジェクトは何でも構いません。また、複数のシステムコールを使って上記２つの要件を満足しても構いませんし、必要であればデータの加工を行っても構いません。

なお、adi_hdr割込みハンドラではA/D変換結果をヘッダファイルに定義されているマクロ名「AD0.ADDRA」で参照し、ad_tskタスクはカーネルオブジェクトで受け取ったA/D変換結果を変数volumeに代入し、以降の処理に進むものとします。

LED表示データの受け渡しに利用するカーネルオブジェクト



LED表示データの受け渡しに利用するカーネルオブジェクトの要件

７セグメントLEDへの表示データ（４バイト）はad_tskタスクが作成し、led_tskタスクに何らかのカーネルオブジェクトを使って受け渡します。データの受け渡しが行えるカーネルオブジェクトであれば、何でも構いませんが以下に示す２つの要件があります。

- LED表示データの上書きに対する対策が行えるカーネルオブジェクトであること
- LED表示データが存在しなくても、led_tskタスクは処理の継続が行えること

１つ目の要件はA/D変換速度と７セグメントLEDの表示速度の関係です。A/D変換速度は1.04μsで行えるのに対して、７セグメントLEDはダイナミック点灯が必要なことから20msで行います。つまり、led_tskタスクが表示データを受信する速度より、ad_tsk タスクが表示データを送信する速度の方が圧倒的に速いのです。結果、LED表示データの上書きに対する対策が行えるカーネルオブジェクト、ないしはLED表示データの受け渡しとは別のカーネルオブジェクトを利用して、LED表示データの上書きに対する対策を行う必要があります。

また、A/D変換はIRQ15割込みによって開始・停止を切り替え可能です。このため、LED表示データは必ずしも選択したカーネルオブジェクト内に存在するとは限りません。一方、LED 表示データが存在しなくても、７セグメントLEDのダイナミック点灯は継続して行う必要があります。つまり、led_tskタスクはLED表示データが存在しなくてもWAITING状態となってはなりませんから、ポーリング機能を使う必要があります。

なお、ad_tskタスクは変数dataに格納されているLED表示データを選択したカーネルオブジェクトに送信します。led_tskタスクは選択したカーネルオブジェクトから受信したLED表示データを変数dataに格納し、以降の処理に進むものとします。

4.2　プログラムリスト

irq15_hdr（割込みハンドラ：割込み優先レベル２）

```
79: /*****************************************************************************/
80: /*  IRQ15 Interrupt Handler                                                  */
81: /*****************************************************************************/
82: EXPORT void irq15_hdr(UINT dintno)
83: {
84:     IEN( AD0, ADI0 ) ^= 1;                  // Change ADI0 Request
85: }
```

解説

82～85行目： 可変ベクタ番号79番のIRQ15割込み対応のirq15_hdr割込みハンドラです。
84行目： ADI0のIEN（A/D変換器の割込み許可ビット）を反転します。

adi_hdr（割込みハンドラ：割込み優先レベル１）

```
87: /*****************************************************************************/
88: /*  ADI0 Interrupt Handler                                                   */
89: /*****************************************************************************/
90: EXPORT void adi_hdr(UINT dintno)
91: {
92:     setpsw_i( );                            // Enable Mutiply Interrupt
93: // カーネルオブジェクトを利用し、AD0.ADDRA(10bit)をad_tskに渡す
94: // 制約事項①割込みハンドラから設定が可能なカーネルオブジェクトを利用
95: //         ②A/D変換結果には値ゼロが存在する
96: }
```

解説

90～96行目： 可変ベクタ番号98番のADI0割込み対応のadi_hdr割込みハンドラです。
93～95行目： カーネルオブジェクトを利用し、A/D変換結果（マクロ名：AD0.ADDRA）をad_tskに渡します。

ad_tsk（タスク）

```
 98: /*************************************************************************/
 99: /*  A/D Task                                                             */
100: /*************************************************************************/
101: EXPORT void ad_tsk(INT stacd, VP exinf)
102: {
103: static const char seg[] =                    // Segment Data
104:     { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x27, 0x7F, 0x6F };
105: // Return Parameter
106: UW data;                                     // Parameter
107: int volume, i;                               // Volume, Loop Counter
108:
109:     while( 1 ) {
110:         AD0.ADCSR.BIT.ADST = 1;             // A/D Start
111: // カーネルオブジェクトを利用し、ad_hdrよりA/D変換結果を受け取る
112: // その結果を変数volumeに格納する
113: // 制約事項①割込みハンドラから設定が可能なカーネルオブジェクトを利用
114: //         ②A/D変換結果には値ゼロが存在する
115:         for( i=0 ; i<4 ; i++ ) {            // Digit Loop
116:             ((char *)&data)[i] = seg[volume%10];    // Set Segment Line
117:             if( !(volume /= 10) )           // Set Next Analize
118:                 break;                      // Zero Find ?
119:         }
120:         while( ++i<4 )                      // Digit Loop
121:             ((char *)&data)[i] = 0;         // Clear Segment Line
122: // カーネルオブジェクトを利用し、LED表示データ(変数data)をled_tskに渡す
123: // 制約事項①A/D変換速度とLED表示速度では、A/D変換速度の方が速い
124: //           つまり、LEDへの表示データの上書き対策が必要
125:     }
126: }
```

解説

101～126行目： A/D変換結果からLED表示データを作成するad_tskタスクです。
105行目： 必要であれば、システムコールで使用するリターンパラメータを宣言します。
110行目： ADCSRのADSTビットをセットし、A/D変換を開始します。
111～114行目： カーネルオブジェクトを利用し、A/D変換結果をadi_hdr割込みハンドラより受け取り、変数volumeに格納します。
115～121行目： A/D変換結果からLED表示データを作成します。
122～124行目： カーネルオブジェクトを利用し、LED表示データである変数dataをled_tskタスクに渡します。

led_tsk（タスク）

```
128: /*************************************************************************/
129: /*  LED Task                                                             */
130: /*************************************************************************/
131: EXPORT void led_tsk(INT stacd, VP exinf)
132: {
133: static const unsigned char led[] = { 0x10, 0x20, 0x40, 0x80 };
134: UW data=0;                                    // Return Parameter
135: int index;                                    // Array Index
136:
137:     while( 1 ) {
138:         for( index=0 ; index<4 ; index++ ) {// Digit Loop
139:             PORT4.DR.BYTE = 0x00;           // Disable Digit Line
140:             PORTE.DR.BYTE = ((char *)&data)[index];    // Set Segment Line
141:             PORT4.DR.BYTE = led[index];     // Enable Digit Line
142:             tk_slp_tsk( TMO_FEVR );         // Wait 5ms
143:         }
144: // カーネルオブジェクトを利用し、LED表示データを受け取り、変数dataに格納
145: // 制約事項①LEDへの表示データは、必ず存在するとは限らない
146: //         ②表示データが無い場合、その状態(変換停止)をLEDに表示すること
147:     }
148: }
```

解説

131～148行目： LED表示データを７セグメントLEDにダイナミック点灯するled_tskタスクです。
138～143行目： tk_slp_tskを使って5msの間隔で７セグメントLEDをダイナミック点灯します。
144～146行目： カーネルオブジェクトを利用し、LED表示データをad_tskタスクより受け取ります。

led_cyc（周期ハンドラ）

```
150: /*************************************************************************/
151: /*  LED Cyclic Handler                                                   */
152: /*************************************************************************/
153: EXPORT void led_cyc(VP exinf)                // 5ms Cyclic
154: {
155:     tk_wup_tsk( ObjID[LED_TSK] );           // Wakeup to led_tsk
156: }
```

解説

153～156行目： led_tskタスクの周期的な起動を行うled_cyc周期ハンドラです。
155行目： tk_wup_tskシステムコールをled_tskタスクに対して発行します。

usermain関数

```
 1: #include <tk/tkernel.h>
 2: #include <tm/tmonitor.h>
 3: #include <machine.h>
 4: #include "iodefine.h"
 5:
 6: EXPORT void irq15_hdr(UINT dintno);
 7: EXPORT void adi_hdr(UINT dintno);
 8: EXPORT void ad_tsk(INT stacd, VP exinf);
 9: EXPORT void led_tsk(INT stacd, VP exinf);
10: EXPORT void led_cyc(VP exinf);
11:
12: typedef enum { AD_TSK, LED_TSK, LED_CYC, OBJ_KIND_NUM } OBJ_KIND;
13: // 使用するカーネルオブジェクトのマクロ名の宣言を追加
14: EXPORT ID ObjID[OBJ_KIND_NUM];                // ID Table
15: /*======================================================================*/
16: /*  usermain                                                            */
17: /*======================================================================*/
18: EXPORT INT usermain( void )
19: {
20: union { T_CTSK ctsk; T_DINT dint; T_CCYC ccyc; } t;
21: // 使用するカーネルオブジェクトのパラメータパケットの宣言を追加
22: ID objid;
23:
24:     t.ctsk.tskatr = TA_HLNG;
25:     t.ctsk.stksz = 1024;
26:     t.ctsk.task =  ad_tsk;                  // Set Address of ad_tsk
27:     t.ctsk.itskpri = 2;                     // Set Priority of ad_tsk
28:     if( (objid = tk_cre_tsk( &t.ctsk )) <= E_OK )
29:         goto ERROR;                         // Create ad_tsk
30:     ObjID[AD_TSK] = objid;                  // Set ID of ad_tsk
31:
32:     t.ctsk.task = led_tsk;                  // Set Address of led_tsk
33:     t.ctsk.itskpri = 1;                     // Set Priority of led_tsk
34:     if( (objid = tk_cre_tsk( &t.ctsk )) <= E_OK )
35:         goto ERROR;                         // Create led_tsk
36:     ObjID[LED_TSK] = objid;                 // Set ID of led_tsk
37:
38:     t.dint.intatr = TA_HLNG;
39:     t.dint.inthdr = irq15_hdr;
40:     if( tk_def_int( VECT( ICU, IRQ15 ), &t.dint ) != E_OK )
41:         goto ERROR;                         // Define irq15_hdr
42:
43:     t.dint.inthdr = adi_hdr;
44:     if( tk_def_int( VECT( AD0, ADI0 ), &t.dint ) != E_OK )
45:         goto ERROR;                         // Define adi_hdr
```

解説

12～14行目：使用するカーネルオブジェクトのID番号に対応したマクロ名と外部変数のobjID配列の宣言です。

18～78行目：システムの初期化を行うusermain関数です。

20～22行目：カーネルオブジェクトを生成する際に使用するパラメータパケットを共用体変数tとして宣言します。

```
47:     t.ccyc.cycatr = TA_HLNG | TA_STA;
48:     t.ccyc.cychdr = led_cyc;
49:     t.ccyc.cyctim = 5;
50:     t.ccyc.cycphs = 500;
51:     if( (objid = tk_cre_cyc( &t.ccyc )) <= E_OK )
52:         goto ERROR;                         // Create led_cyc
53:     ObjID[LED_CYC] = objid;                 // Set ID of led_cyc
54:
55: // 使用するカーネルオブジェクトを生成する
56:
57:     PORTE.DDR.BYTE = 0xFF;                  // Enable Segment Line
58:     PORT4.DDR.BYTE = 0xF0;                  // Enable 7 Seg.LED
59:
60:     PORT0.ICR.BIT.B7 = 1;                   // Enable IRQ15 Input Buffer
61:     ICU.IRQCR[15].BIT.IRQMD = 1;            // Set falling Edge IRQ15
62:     IR( ICU, IRQ15 ) = 0;                   // Clear IRQ15 Interrupt
63:     IPR( ICU, IRQ15 ) = 2;                  // IRQ15 Interrupt Level is 2
64:     IEN( ICU, IRQ15 ) = 1;                  // Enable IRQ15 Interrupt
65:
66:     MSTP( AD0 ) = 0;                        // Wakeup A/D0
67:     AD0.ADCSR.BIT.ADIE = 1;                 // Enable ADI0 Request
68:     IPR( AD0, ADI0 ) = 1;                   // ADI0 Interrupt Level is 1
69:     IEN( AD0, ADI0 ) = 1;                   // Enable ADI0 Interrupt
70:
71:     tk_sta_tsk( ObjID[AD_TSK], 0 );         // Start ad_tsk
72:     tk_sta_tsk( ObjID[LED_TSK], 0 );        // Start led_tsk
73:
74:     while( 1 )  wait( );
75: ERROR:
76:     return 0;
77: }
```

解説

24～45行目： ad_tskタスク、led_tskタスクの生成と、irq15_hdr割り込みハンドラ、adi_hdr割り込みハンドラの定義します。

47～53行目： led_tskタスクを周期的に起床するled_cyc周期ハンドラを生成します。

57～58行目： ７セグメントLEDが接続されているポートを初期化します。

60～64行目： IRQ15を割り込み優先レベル２で初期化します。

66～69行目： A/D変換器チャネル０を割り込み優先レベル１で初期化します。

71～72行目： ad_tskタスクとled_tskタスクを起動します。

4.3　解答例

usermain関数

```
 1: #include <tk/tkernel.h>
 2: #include <tm/tmonitor.h>
 3: #include <machine.h>
 4: #include "iodefine.h"
 5:
 6: EXPORT void irq15_hdr(UINT dintno);
 7: EXPORT void adi_hdr(UINT dintno);
 8: EXPORT void ad_tsk(INT stacd, VP exinf);
 9: EXPORT void led_tsk(INT stacd, VP exinf);
10: EXPORT void led_cyc(VP exinf);
11:
12: typedef enum { AD_TSK, LED_TSK, LED_CYC, AD_FLG, LED_MBF, OBJ_KIND_NUM }
                                                                    OBJ_KIND;
13: EXPORT ID ObjID[OBJ_KIND_NUM];              // ID Table
14:
15: /*=========================================================================*/
16: /*  usermain                                                               */
17: /*=========================================================================*/
18: EXPORT INT usermain( void )
19: {
20: union { T_CTSK ctsk; T_DINT dint; T_CCYC ccyc; T_CFLG cflg; T_CMBF cmbf; }
                                                                          t;
21: ID objid;
22:
23:     t.ctsk.tskatr = TA_HLNG;
24:     t.ctsk.stksz = 1024;
25:     t.ctsk.task =  ad_tsk;                  // Set Address of ad_tsk
26:     t.ctsk.itskpri = 2;                     // Set Priority of ad_tsk
27:     if( (objid = tk_cre_tsk( &t.ctsk )) <= E_OK )
28:         goto ERROR;                         // Create ad_tsk
29:     ObjID[AD_TSK] = objid;                  // Set ID of ad_tsk
30:
31:     t.ctsk.task = led_tsk;                  // Set Address of led_tsk
32:     t.ctsk.itskpri = 1;                     // Set Priority of led_tsk
33:     if( (objid = tk_cre_tsk( &t.ctsk )) <= E_OK )
34:         goto ERROR;                         // Create led_tsk
35:     ObjID[LED_TSK] = objid;                 // Set ID of led_tsk
36:
37:     t.dint.intatr = TA_HLNG;
38:     t.dint.inthdr = irq15_hdr;
39:     if( tk_def_int( VECT( ICU, IRQ15 ), &t.dint ) != E_OK )
40:         goto ERROR;                         // Define irq15_hdr
41:
42:     t.dint.inthdr = adi_hdr;
43:     if( tk_def_int( VECT( AD0, ADI0 ), &t.dint ) != E_OK )
44:         goto ERROR;                         // Define adi_hdr
```

解説

12行目：イベントフラグとメッセージバッファのID番号に対応した配列の添字のマクロ名を宣言します。

20行目：イベントフラグとメッセージバッファの生成に利用するパラメータパケットのメンバを宣言します。

```
46:     t.ccyc.cycatr = TA_HLNG | TA_STA;
47:     t.ccyc.cychdr = led_cyc;
48:     t.ccyc.cyctim = 5;
49:     t.ccyc.cycphs = 500;
50:     if( (objid = tk_cre_cyc( &t.ccyc )) <= E_OK )
51:         goto ERROR;                         // Create led_cyc
52:     ObjID[LED_CYC] = objid;                 // Set ID of led_cyc
53:
54:     t.cflg.flgatr = TA_TFIFO | TA_WSGL;
55:     t.cflg.iflgptn = 0;                     // Set Initial Pattern
56:     if( (objid = tk_cre_flg( &t.cflg )) <= E_OK )
57:         goto ERROR;                         // Create ad_flg
58:     ObjID[AD_FLG] = objid;                  // Set ID of ad_flg
59:
60:     t.cmbf.mbfatr = TA_TFIFO;
61:     t.cmbf.bufsz = 0;                       // Set Buffer Size
62:     t.cmbf.maxmsz = 4;                      // Set Maximun Size
63:     if( (objid = tk_cre_mbf( &t.cmbf )) <= E_OK )
64:         goto ERROR;                         // Create led_mbf
65:     ObjID[LED_MBF] = objid;                 // Set ID of led_mbf
66:
67:     PORTE.DDR.BYTE = 0xFF;                  // Enable Segment Line
68:     PORT4.DDR.BYTE = 0xF0;                  // Enable 7 Seg.LED
69:
70:     PORT0.ICR.BIT.B7 = 1;                   // Enable IRQ15 Input Buffer
71:     ICU.IRQCR[15].BIT.IRQMD = 1;            // Set falling Edge IRQ15
72:     IR( ICU, IRQ15 ) = 0;                   // Clear IRQ15 Interrupt
73:     IPR( ICU, IRQ15 ) = 2;                  // IRQ15 Interrupt Level is 2
74:     IEN( ICU, IRQ15 ) = 1;                  // Enable IRQ15 Interrupt
75:
76:     MSTP( AD0 ) = 0;                        // Wakeup A/D0
77:     AD0.ADCSR.BIT.ADIE = 1;                 // Enable ADI0 Request
78:     IPR( AD0, ADI0 ) = 1;                   // ADI0 Interrupt Level is 1
79:     IEN( AD0, ADI0 ) = 1;                   // Enable ADI0 Interrupt
80:
81:     tk_sta_tsk( ObjID[AD_TSK], 0 );         // Start ad_tsk
82:     tk_sta_tsk( ObjID[LED_TSK], 0 );        // Start led_tsk
83:
84:     while( 1 )  wait( );
85: ERROR:
86:     return 0;
87: }
```

解説

54～58行目：adi_hdr割り込みハンドラとad_tskタスクの間で利用するイベントフラグを生成します。イベントフラグの初期値はゼロ、待ちに入るタスクはad_tskタスクだけであるため、属性としてTA_TFIFOとTA_WSGLを指定（片方でも大丈夫）します。

60～65行目：ad_tskタスクとled_tskタスクの間で利用するメッセージバッファを生成します。送信側と受信側で同期を取るためにメッセージバッファのサイズはゼロ、メッセージの最大長は４バイト、属性としてTA_TFIFOを指定します。

adi_inh（割込みハンドラ）

```
 97: /*************************************************************************/
 98: /*  ADI0 Interrupt Handler                                               */
 99: /*************************************************************************/
100: EXPORT void adi_hdr(UINT dintno)
101: {
102:     setpsw_i( );                                  // Enable Mutiply Interrupt
103:     tk_set_flg( ObjID[AD_FLG], AD0.ADDRA+1 );       // set A/D result
104: }
```

解説

103行目： tk_set_flgシステムコールでA/D変換結果をad_tskタスクに渡します。なお、変換結果ゼロに対応するため、変換結果に+1した値をsetptnに指定します。

ad_tsk（タスク）

```
106: /*************************************************************************/
107: /*  A/D Task                                                             */
108: /*************************************************************************/
109: EXPORT void ad_tsk(INT stacd, VP exinf)
110: {
111: static const char seg[] =                       // Segment Data
112:     { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x27, 0x7F, 0x6F };
113: UINT flgptn;                                    // Return Parameter
114: UW data;                                        // Parameter
115: int volume, i;                                  // Volume, Loop Counter
116:
117:     while( 1 ) {
118:         AD0.ADCSR.BIT.ADST = 1;                 // A/D Start
119:         tk_wai_flg( ObjID[AD_FLG], 0x7FF, TWF_ORW | TWF_CLR, &flgptn,
120:                                                     TMO_FEVR ); // Wait A/D End
121:         volume = flgptn - 1;                    // Adjust A/D Result
122:         for( i=0 ; i<4 ; i++ ) {                // Digit Loop
123:             ((char *)&data)[i] = seg[volume%10];    // Set Segment Line
124:             if( !(volume /= 10) )               // Set Next Analize
125:                 break;                          // Zero Find ?
126:         }
127:         while( ++i<4 )                          // Digit Loop
128:             ((char *)&data)[i] = 0;             // Clear Segment Line
129:         tk_snd_mbf( ObjID[LED_MBF], &data, sizeof(data), TMO_FEVR );
130:     }
131: }
```

解説

119～120行目： tk_wai_flgシステムコールでA/D変換結果をadi_hdr割込みハンドラより受け取ります。なお、変換結果は10ビットですが、変換結果に+1しているため、下位11ビットをwaiptn、待ちモードにはOR待ちとクリアを指定します。

121行目： tk_wai_flgシステムコールの待ち解除後は、受け取った値を-1して変換結果を補正します。

129行目： tk_snd_mbfシステムコールでLED表示データをled_tskに送信します。

led_tsk（タスク）

```
133: /*****************************************************************************/
134: /*  LED Task                                                                 */
135: /*****************************************************************************/
136: EXPORT void led_tsk(INT stacd, VP exinf)
137: {
138: static const unsigned char led[] = { 0x10, 0x20, 0x40, 0x80 };
139: UW data=0;                                    // Return Parameter
140: int index;                                    // Array Index
141:
142:     while( 1 ) {
143:         for( index=0 ; index<4 ; index++ ) {// Digit Loop
144:             PORT4.DR.BYTE = 0x00;           // Disable Digit Line
145:             PORTE.DR.BYTE = ((char *)&data)[index];    // Set Segment Line
146:             PORT4.DR.BYTE = led[index];     // Enable Digit Line
147:             tk_slp_tsk( TMO_FEVR );         // Wait 5ms
148:         }
149:         tk_rcv_mbf( ObjID[LED_MBF], &data, TMO_POL );   // Receive Segment
150:     }
151: }
```

解説

149行目： tk_rcv_mbfシステムコールでLED表示データをad_tskタスクより受け取ります。ただし、LED表示データが存在しなくてもWAITING状態とならないようにTMO_POLのポーリングを指定します。

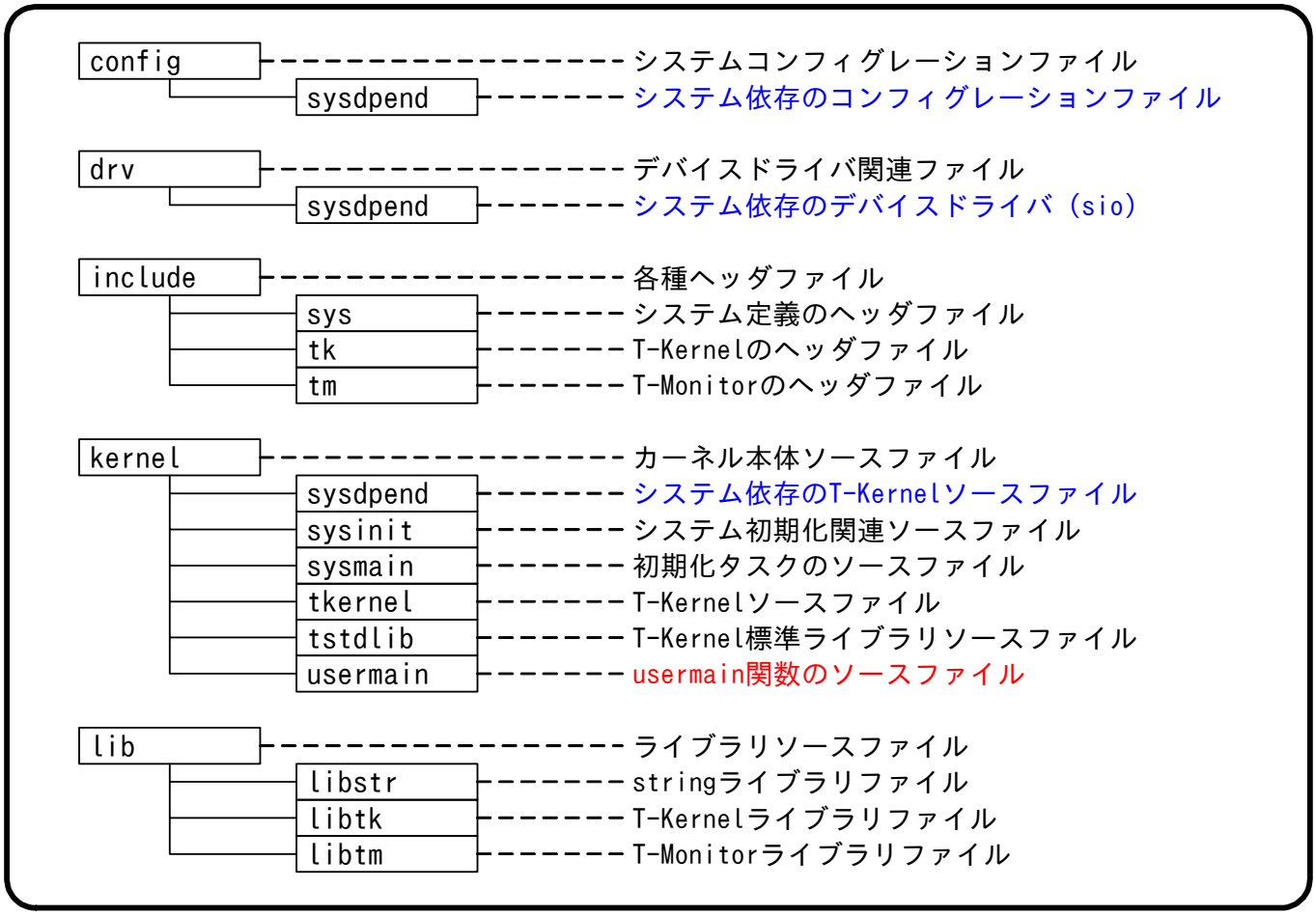
メモ

第 5 章

リファレンスコードの概要

5.1　リファレンスコードの概要

リファレンスコードのファイル構成



リファレンスコードのファイル構成

小規模組み込みシステムでは最適化・適応化が特に重要であることから、μT-Kernel は T-Kernel とは異なり、ソースコードの単一性を保持していません。その代わり、リファレンスコードと呼ばれる参照用のソースコードが提供されています。

リファレンスコードは μT-Kernel の実装の一つであり、T-Engine Forum により配布されています。T-Kernel と異なっているのは、このリファレンスコードだけが μT-Kernel であるということではなく、OS 実装者はこれを改造しても良いし、全く独自の実装をしても良いことです。ただし、リファレンスコードと同じ振舞いをするもののみが μT-Kernel 仕様 OS として認められます。

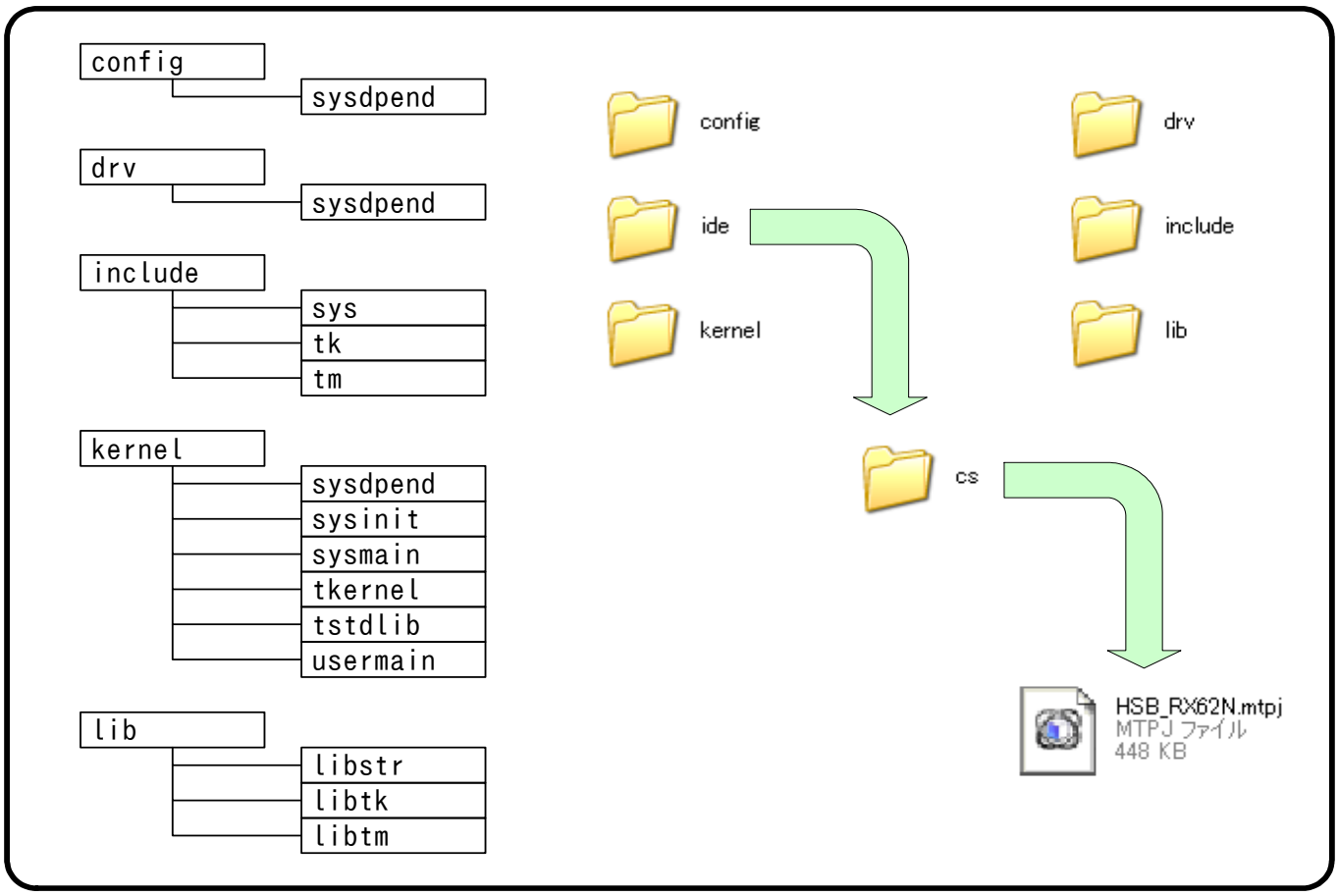
リファレンスコードは仕様書だけでは規定するのが難しい細かい部分の動作までを規定するものであり、このリファレンスコードという概念の導入により、ターゲットに合わせた適応化・最適化を行ないやすくしつつも、異なる実装に対して動作の一貫性を保つことができるようになっています。

そのリファレンスコードの主なファイル構造は上記の通りです。

- config　‥ システムのコンフィグレーションファイル
- drv …… デバイスドライバ関連ファイル
- include ‥ ヘッダファイル
- kernel　‥ カーネル本体ソースファイル
- lib …… ライブラリソースファイル

なお、本セミナーで使用しない一部のソースファイルやフォルダは記載していませんので注意してください。

フォルダ構成



フォルダ構成

μT-Kernel のフォルダ構成は、前頁のファイル構成と殆ど同じです。ただし、IDE 関連の格納フォルダとして ide が存在します。今回の提供する μT-Kernel は CubeSuite+ 用にカスタマイズされたものであり、cs フォルダ内にある HSB_RX62N のアイコンから起動します。

なお、ユーザが作成したソースファイルはどこに配置しても構いませんが、デフォルトでは cs フォルダが参照元となり、ビルド後のロードモジュールも cs フォルダ内のコンフィグレーション・フォルダに生成されます。

5.2　ユーザシステムの起動

usermain関数の実行

kernel/sysmain/src/inittask_main.c

```
 1: #include ”sysmain.h”
 2: #include ”kernel.h”
 3: #include <sys/debug.h>
 4:
 5: LOCAL const char knl_boot_message[] = {    /* Boot message */
 6:     BOOT_MESSAGE
 7: };
 8: /*
 9:  * Initial task Main
10:  *    The available stack size is slightly less than 8KB.
11:  *    The initial task is the system task,
12:  *    so it should not be deleted.
13:  */
14: EXPORT INT knl_init_task_main( void )
15: {
16:     INT    fin;
17:     tm_putstring((UB*)knl_boot_message);
18:     fin = 1;
19:     if ( fin > 0 ) {
20:         /* Perform user main */
21:         fin = usermain( );
22:     }
23:     return fin;
24: }
```

解説

14～24行目： カーネルの初期タスクです。

16行目： 起動ステータスを示す変数finを宣言します。

17行目： T-Monitorのライブラリ関数を使用し、ブートメッセージをコンソールに表示します。

18～22行目： 起動ステータスを設定しながら、useermain関数を実行します。

23行目： 起動ステータスをreturn文で返します。

usermain関数の実行

usermain関数は上記のリストが示す通り、初期タスクのコンテキストで実行されます。タスク優先度は最大優先度から2を減算した値(CFN_MAX_PRI-2)がデフォルトとなっています。

なお、初期タスクの本来のリストでは、ユーザ初期化プログラム（ミドルウェアやデバイスドライバ）の起動および停止処理が前後に存在しますが、説明の都合上省略しています。

usermain関数

kernel/usermain/usermain.c

```
 1: #include <basic.h>
 2: #include <tk/tkernel.h>
 3: #include <tm/tmonitor.h>
 4:
 5: /*
 6:  * Entry routine for the user application.
 7:  * At this point, Initialize and start the user application.
 8:  *
 9:  * Entry routine is called from the initial task for Kernel,
10:  * so system call for stopping the task should not be issued
11:  * from the contexts of entry routine.
12:  * We recommend that:
13:  * (1)'usermain()' only generates the user initial task.
14:  * (2)initialize and start the user application by the user
15:  * initial task.
16:  */
17:
18: EXPORT INT usermain( void )
19: {
20:     tm_putstring((UB*)"Push any key to shutdown the micro T-Kernel.¥n");
21:     tm_getchar(-1);
22:
23:     return 0;
24: }
```

解説

18～24行目： ユーザシステムのエントリ関数であるusermain関数です。
20行目： T-Monitorのライブラリ関数を使用し、メッセージを表示します。
21行目： T-Monitorのライブラリ関数を使用し、キーボードから１文字入力します。
23行目： 正常終了であるゼロをreturn文で返します。

usermain関数

上記のusermain関数がユーザシステムのエントリ関数となります。ここからユーザシステムの初期化タスクを生成、起動します。なお、usermain関数が終了するとシステムはシャットダウンに入ります。従いまして、電源オフまで終了しないシステムの場合、usermain関数は無限ループする必要があります。また、usermain関数は初期タスクの優先度で動作します。もし、初期タスクの優先度を変更しない場合、usermain関数は何らかのシステムコールによってWAITING状態とならなければ、初期タスクより低い優先度のタスクは動作しません。

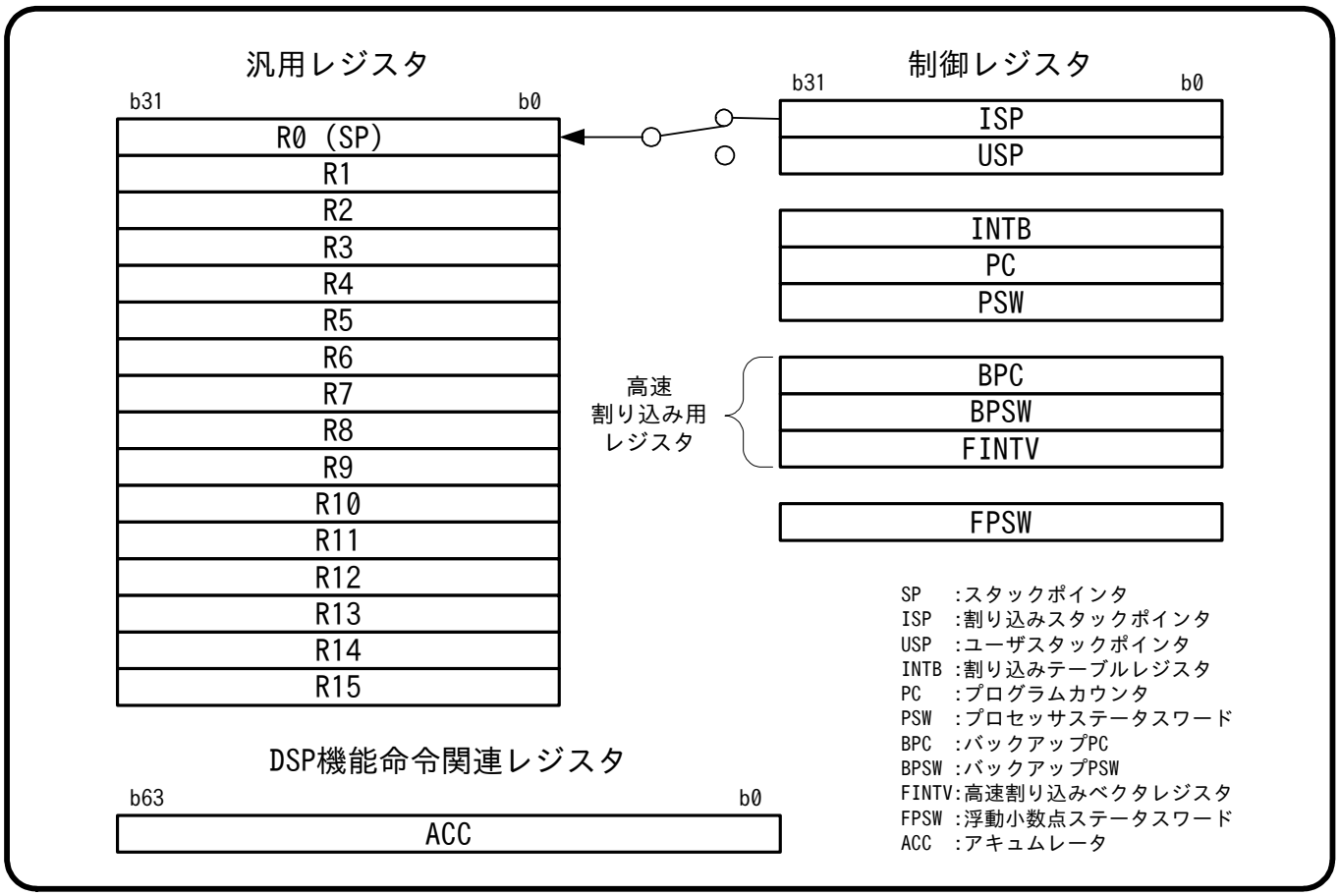
メモ

付録 1

RX600 シリーズの概要

付録 1.1　RX600 シリーズの CPU 内部レジスタ

CPU内部レジスタ



PSW（プロセッサステータスワード）

PSW（プロセッサステータスワード）

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	IPL				—	—	—	PM	—	—	U	I
初期値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	—	—	—	O	S	Z	C
初期値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

IPLビット（プロセッサ割り込み優先レベル）
　プロセッサ割り込み優先レベルを指定します。値が大きい程、受け付けを禁止している優先レベルが高くなります。
　なお、RX610の場合、IPLビットの最上位ビットは無効です。
PMビット（プロセッサモード設定ビット）
　プロセッサモードを設定するビットです。”0”でスーパーバイザモード、”1”でユーザモードになります。
　例外を受け付けると ”0”のスーパバイザモードになります。
Uビット（スタックポインタ指定ビット）
　使用するスタックポインタ（ISP/USP）を指定するビットです。”0”でISP、”1”でUSPを使用になります。
　例外を受け付けると ”0”のISPを使用になります。
　スーパバイザモードからユーザモードに移行すると ”1”のUSPを使用になります。
Iビット（割り込み許可ビット）
　割り込み要求の受け付けを許可するビットです。”0”で割り込み禁止、”1”で割り込み許可になります。
　例外を受け付けると ”0”の禁止状態になります。

FPSW（浮動小数点ステータスワード）

FPSW（浮動小数点ステータスワード）

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	FS	FX	FU	FZ	FO	FV	—	—	—	—	—	—	—	—	—	—
初期値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	EX	EU	EZ	EO	EV	—	DN	CE	CX	CU	CZ	CO	CV	RM	
初期値	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/(W)	R/(W)	R/(W)	R/(W)	R/(W)	R/(W)	R/W	R/W

DNビット（非正規化数の0フラッシュビット）
　”0”のとき非正規化数を非正規化数として扱います。”1”のとき非正規化数をゼロとして扱います。
RMビット（浮動小数点丸めモード設定ビット）
　浮動小数点丸めモードを設定します。
【浮動小数点丸めモードの説明】
・最近値への丸め（デフォルト）：無限の有効桁を持つとして計算した場合の結果と近い方の値へ丸める
　　中間時は結果が偶数になる方向へ丸める
・0 方向への丸め：結果の絶対値が小さくなる方向へ丸める（単純な切り捨て）
・＋∞ 8 方向への丸め：結果の値が大きくなる方向へ丸める
・－∞方向への丸め：結果の値が小さくなる方向へ丸める
（1）「最近値への丸め」はデフォルトのモードであり、最も正確な値を返します。
（2）「0 方向への丸め」、「＋∞方向への丸め」、「－∞方向への丸め」は、区間演算（Interval arithmetic）を使用した精度保証を行うときに使用します。

トロンフォーラム
【実習】μT-Kernel 入門(協力:ルネサス エレクトロニクス)

2016 年 6 月 23 日発行

発行所
トロンフォーラム
(YRP ユビキタス・ネットワーキング研究所内)
〒141-0031 東京都品川区西五反田 2-12-3 第一誠実ビル 9F
URL: http://www.tron.org/ja/
TEL:03-5437-0572(代表) FAX:03-5437-2399(代表)

トロンフォーラム©2016
Printed in Japan